# THE BBC MICRO BOOK

BASIC, Sound and Graphics

JIM McGREGOR & ALAN WATT

The BBC Micro Book
BASIC, Sound and Graphics

JIM McGREGOR

ALAN WATT

Contents

Preface

The first half of this book is an introduction to programming in BASIC for the BBC micro. The second half of the book is a comprehensive guide to the more advanced features of this powerful computer. The book can be used by complete beginners or by anyone who already has some experience in BASIC programming on other machines.

BBC BASIC contains many features not usually found in microcomputer BASIC and these are exploited extensively in the book. Structured programming is introduced at an early stage and used throughout. In this respect the book is emphatically not another standard BASIC book re-hashed, but has been written specially for the BBC machine. Well over half the book (Chapter 8 onwards) deals exclusively with Teletext, graphics, animation and sound. In addition, elementary use of sound and graphics appears in earlier chapters.

Throughout the book general ideas are introduced by means of example programs. Considerable care has been taken to make sure that the programs are readable and they can be viewed almost as part of the text rather than a tangle that needs careful unravelling. There is a large number of suggested exercises because in the end programming is a practical art that can only be acquired by doing it.

The book is meant to be self contained and although the User Guide is always a necessary reference, most of the material that you will require for day to day programming is contained herein. Some of the more detailed technical information is to be found in the ten appendices at the end of the book.

A mastery of the material in this book will mean that you have acquired a good deal of expertise and your programming ability should be limited only by your imagination. Too often people stop programming at a certain level of complexity because they see the work involved as being too difficult. Often 'too difficult' means 'too tedious and detailed'. Using the structured programming techniques described herein should help to remove this artificial block.

Are you a slow typist?

You can experiment with the programs described in this book
without having to type them. All the programs of any length
are available on a computer cassette

  The BBC Micro Tape: BASIC, Sound and Graphics
            ISBN 0-201-14510-3

Some of the longer programs on the cassette are

  Stock control (Chapters 6 and 7)
  Teletext design (Chapter 8)
  Composite images and image planes (Chapter 9)
  Rubber banding and picking and dragging (Chapter 9)
  Playing tunes (Chapter 10)
  The keyboard as an organ (Chapter 10)
  Animated games (Chapter 11)
  Character design program (Chapter 11)
  Spinning discs and spirals (Chapter 11)

Programs are also available on cassette for the companion
volume by the same authors:

  Advanced Programming Techniques for the BBC Micro
            ISBN for book 0-201-14059-4
            ISBN for tape 0-201-14512-X
            ISBN for book/tape package 0-201-14513-8

Books and cassettes are available from your bookseller or
direct from

  Addison-Wesley Publishers Limited
  53 Bedford Square
  London WC1B 3DZ

Introduction

In this introductory chapter, we shall use some very simple BBC BASIC programs to introduce a number of general points about using your computer. If you are not familiar with the use of a typewriter keyboard, you should refer to the start of Appendix 1. We assume that you have connected your computer to your television set, switched on and tuned the television if necessary. When you have done this, you should see a display like the following on the screen:

    BBC Computer 16K

    BASIC

    >

The symbol '>' is displayed by the computer whenever it is waiting for you to tell it what to do next. Try typing

    PRINT 43 + 26

You must press the RETURN key at the end of every line that you type. The computer will not pay any attention to anything you type until you press this key. If you make a mistake on the line before you press the RETURN key, you can change the line by using the DELETE key to delete as many characters as you like. When you press the RETURN key at the end of the above line, the computer should immediately 'print'

    69

on the screen. We have given the computer a single BASIC instruction and it has been obeyed immediately - the computer can be used like a rather sophisticated calculator. Instructions written in BASIC are often referred to as 'statements'.

Storing a program

The big advantage that a computer has over a calculator is that a list of instructions can be stored inside the computer. Such a list of instructions is called a 'program'

and, once stored in the computer, the instructions in a
program can be obeyed again and again. If we want the
computer to store a line of BASIC as part of a program, we
must number the line. The lines in a program are stored in
the order determined by these numbers. Try typing:

    1 PRINT 43 + 26

Nothing will happen, but the computer has stored this
instruction as line 1 of a program. Now type:

    2 PRINT 43 * 26

(The '*' is used as a multiplication sign to avoid confusion
with the letter 'x'.) The computer now has two lines stored
away. In order to convince yourself that the computer has
stored these lines, type the command:

    LIST

and the computer will display a list of the lines in your
program so far. If you want the computer to obey the
instructions in your program you must tell it to 'run' the
program by typing the command:

    RUN

This will make the computer obey the statements in the
program one after another in the order in which they are
stored. In this case, the computer will print:

        69
        1118

The lines in a program do not need to be numbered from 1
upwards, nor do the numbers need to go up in steps of 1.
Type the command:

    NEW

to tell the computer to erase the old program in order that
we can type a new one. Now type the same program as before
but with different line numbers.

    10   PRINT 43 + 26
    20   PRINT 43 * 26

You can now LIST or RUN the program as before, but the
advantage of numbering the lines in steps of 10 is that new
lines can be easily inserted. For example, type:

```
15  PRINT 43 - 26
```

and LIST the program. You will find that this new line has been inserted between lines 10 and 20. When you run the program, three values are printed on the screen.

You should get into the habit of numbering the lines of all your programs in steps of 10.

Programs and INPUT

We said above that an important feature of a computer is its ability to store a program and run it again and again. Doing this with the program we typed above is not very useful - the same answers are printed every time it is run. When we write a program, we usually describe the operations that the program is to carry out without specifying the actual values to be used in these operations. We do this by giving names to the values involved and describing an operation in terms of these names. Use the NEW command to clear away the old program and type:

```
10  PRINT length*width
```

This BASIC instruction describes the process of multiplying two values to which we have given the names 'length' and 'width'. These names might refer to the length and width of a room for which we wish to calculate the floor area.

Before this program can be successfully run, arrangements must be made to tell the computer what values to use for the variables 'length' and 'width'. Try running the program as it stands and the computer will display the error message:

```
No such variable at line 10
```

This simply means that the computer does not know what values to use in calculating 'length*width'.

There are various ways in which values could be provided for our two variables, but by far the commonest and most important is to tell the computer that we are going to 'input' these values when the program is running. We can do this by adding the following statement to the program:

```
5  INPUT length, width
```

The complete program now consists of two statements:

```
5   INPUT length, width
10  PRINT length*width
```

Now run the program. You will find that a question-mark appears on the screen and nothing further happens. The

computer is trying to obey the statement on line 5 and is waiting for you to INPUT values for 'length' and 'width'. Type two numbers, separated by a comma, for example:

   6, 3

These two values are given to the two variables in the INPUT statement. 'Length' is given the value 6 and 'width' is given the value 3. Only once these two values have been typed can the computer go on and obey the PRINT statement and display the answer:

     18

If you run the same program again, you can input different values and get different answers. This program is much more generally useful than the one we wrote in the last section. The same program can be used for calculating the area of a different room each time it is run.

The use of named variables in a program is fundamental. A program written in terms of names specifies a general task that is carried out using particular values when the program is obeyed. The particular values can be provided via INPUT statements or, as we shall see later, they may be calculated by the program itself from input values.

We shall present more information about doing calculations in the next chapter. In the remaining two sections of this chapter, we shall present simple programs that introduce two of the most exciting features of your BBC micro - its ability to draw pictures and make sounds.

Drawing pictures

The computer graphics facilities on the BBC micro are very sophisticated and techniques for using graphics are extensively illustrated throughout this book. In order to give you a taste of what can be done, here is a short graphics program for you to type into your machine.

```
10   MODE 5
20   MOVE 570, 570
30   DRAW 700, 740  :  DRAW 900, 660
40   DRAW 730, 680  :  DRAW 640, 500
50   DRAW 740, 440  :  DRAW 660, 370
60   DRAW 600, 460  :  DRAW 460, 370
70   DRAW 380, 110  :  DRAW 400, 430
80   DRAW 530, 530  :  DRAW 490, 570
90   DRAW 480, 590  :  DRAW 490, 600
100  DRAW 510, 600  :  DRAW 570, 570
```

When you run this program, you will find that it produces a

line-drawing on your screen.
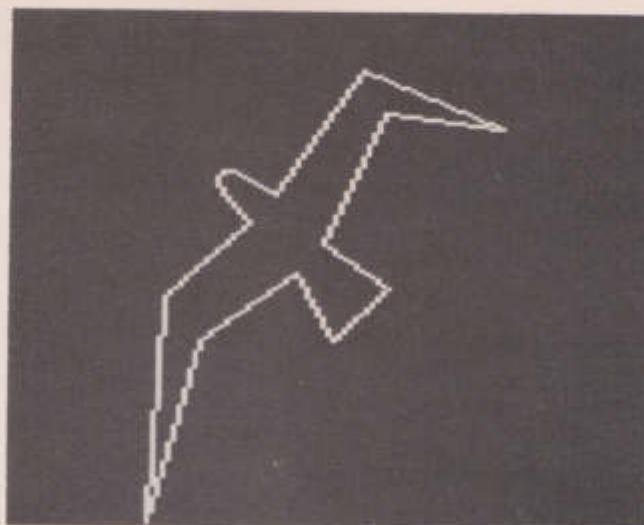
Incidentally we would not normally write such a program by repeating sixteen DRAW statements and better ways of writing such a program are given later. Also note the use of colons to separate statements typed on the same line.

The instruction on line 10 tells the computer to switch to MODE 5. The computer can be in any one of several different 'states' or 'modes' and the MODE 5 instruction switches the computer into one of the modes in which it can draw pictures. After running the above program, the computer remains in MODE 5 and you will find that the characters in this mode are not very readable. You can switch the computer back to its 'normal' mode by typing the instruction

    MODE 7

(without a line number).

When the computer is in one of its graphics modes, we can imagine the screen to be marked out like a piece of graph paper. There are 1279 squares across the screen and 1023 squares up the screen. The instruction:

    MOVE 200,700

moves an imaginary pen to the point that is 200 squares across the screen from the left and 700 squares up the screen.

    DRAW 1000,700

Draws a line from the last point visited to the point that is 1000 squares across and 700 squares up. You should now be able to see how the above program works and you should even be able to write some programs of your own to produce simple line-drawings.

6

Making sounds

Here is a short program that makes use of the sound generator on the BBC micro:

```
10   SOUND 1, -15,   53, 20
20   SOUND 1, -15,   69, 20
30   SOUND 1, -15,   81, 20
```

When you run this program, it plays three notes of a major chord (doh, me, soh), starting on Middle C. The notes are played separately one after another. A detailed explanation of the various parts of the SOUND statement is best left until later, but the third number determines the pitch of of the note played. We can make the chord program more flexible as follows:

```
10   INPUT pitch
20   SOUND 1, -15, pitch, 20
30   SOUND 1, -15, pitch + 16, 20
40   SOUND 1, -15, pitch + 28, 20
```

Try running this program. When the question-mark appears, you should input a number (between 0 and 227). The number you input determines the starting note. If you are musically inclined, you can use this program to work out how the pitch number in a sound statement corresponds to the notes on the piano.

Some general information and advice on typing programs appears in Appendix 1. As your programs get longer and more interesting, you will want to save them on cassette. Information on how to do this appears in Appendix 2.

By now you should be starting to find your way around your computer. Before you can make full use of its potential there is a great deal to learn, so read on!

Chapter 1 **Communicating with your programs**

In this chapter, we describe in more detail various ways of getting information into and out of programs. Input information can consist of numbers and words typed at the keyboard and output information might include numbers, words and pictures displayed on the screen or sounds made by the sound generator. More advanced applications might involve input of signals from an analogue device such as a games paddle or the output of spoken words using a speech synthesizer.

1.1 Input of numeric values

The simple area calculation program of the introductory chapter contained the statements (renumbered)

```
10    INPUT length, width
20    PRINT length*width
```

'Length' and 'width' are the names of two 'variables' that can be given numeric values. In our introductory chapter, we saw how the use of named variables allows us to describe a task in general terms and supply the actual values to be used later when the program is obeyed. It is useful to think of a variable as a named box or 'memory location' in which a program can store a value:

length | 5 | width | 4 |

We say that 'length' is the name of a box containing 5 and 'width' is the name of a box containing 4. Of course the boxes 'length' and 'width' need not necessarily contain the numbers 5 and 4. They can contain any numbers we choose, for example:

length | 3.2 | width | 2.5 |

7

This is why we use the term 'variable'.

The INPUT statement tells the computer that the values are going to be input from the keyboard when the program is running. When the program is run and the INPUT statement is obeyed, the computer will wait until two numbers are typed at the keyboard and these values are then stored in the variables 'length' and 'width':
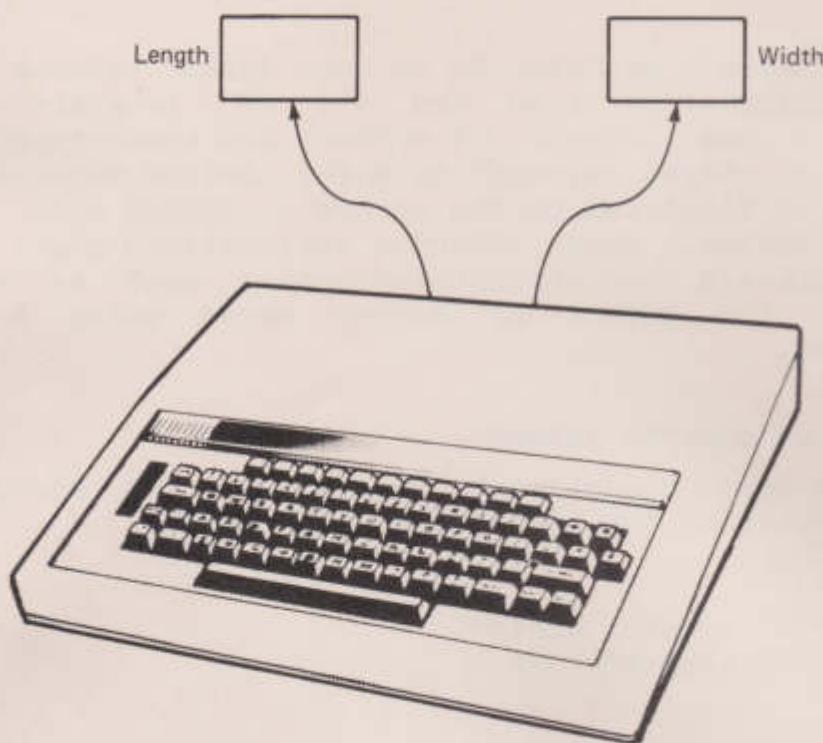
The program then continues running and evaluates the expression

    length*width

using the values for 'length' and 'width' that are contained in the boxes with these names. These are the values that have just been supplied from the keyboard.

When the numbers required by an INPUT statement are being typed at the keyboard, they can all be typed on one line separated from each other by commas, or they can typed on separate lines, the RETURN key being pressed after each number.

When we run the above program, the question mark is all that appears on the screen to remind us that the computer is expecting some input. When a program expects input from the keyboard, it is good programming practice to arrange for it to display a message telling the user of the program what to type. This makes our programs easier for other people to use. We can make our area program do this as follows:

```
10    INPUT "Length of room", length
20    INPUT "Width of room",  width
30    PRINT length*width
```

Each INPUT statement in this amended version of the program
includes a 'string' in double quotation marks before the
variable for which an input value is required. When the
program is run, any string included in an INPUT statement
will be displayed on the screen (without the quotation
marks) before the computer waits for the next value to be
typed. This acts as a 'prompt', telling the user what the
program expects him to type next.

When we run this version of the program, a display such
as the following is produced:

```
>RUN
Length of room?5
Width of room?4
        20
```

The question marks have been inserted automatically by the
computer. If the comma after a string in an INPUT statement
is omitted, then the question mark is suppressed and this is
occasionally desirable.

1.2 Choice of names for numeric variables

In the last section, 'length' and 'width' were used in a
program as the names of two 'variables' in which numbers
were stored by the program when it was run. The programmer
is free to choose the names that he wants to use for his
variables. However, he must not choose a name that can be
confused with a BASIC 'keyword'. PRINT and INPUT are
examples of keywords and words such as these can not be used
as the names of variables. Nor can we use names that start
with BASIC keywords. All the BASIC keywords are typed in
capital letters and the above restriction can be ignored if
we use small letters for the names of all our variables.

You should choose mnemonically meaningful names. For
example in the above context choose the names 'length' and
'width' rather than 'l' and 'w', or, worse still 'a' and
'b'. This makes it much easier for a program to be read and
understood by other people, or by yourself in the future if
you wish to develop it further.

Names can be of any length. Spaces can not be typed in
the middle of a word, but this is not too inconvenient. It
means you must use, for example, 'nooftimes' rather than
'no of times'. However, the underline character can be used
in a variable name and this can be used to distinguish
separate words in a composite name. For example, we could
call the above variable 'no_of_times'.

Finally, if a variable can take only integer (whole

number) values, we can tell the computer this by adding a percent sign, '%', to the end of the variable name. Values stored in integer variables are stored differently from values in other numeric variables and integer arithmetic uses less computer time than real arithmetic. (A 'real' is a number that may have a fractional part.) We shall use integer variables only where speed of program execution is critical, in applications such as animation, which is discussed in Chapter 11.

1.3 More about PRINT

In the last section, a simple PRINT statement was used to tell the computer to calculate the value of an expression and print the result obtained. A print statement can in fact contain a list of several items to be printed. For example, we can make the output produced by the program more informative by using:

    30   PRINT "Area: "; length*width; " square metres."

Incidentally, when you type this line of BASIC in MODE 7 you will find that it will not fit onto one line on the screen. This does not matter. When you get to the right hand edge of the screen, just carry on typing without pressing RETURN and the line you are typing will automatically overflow onto the next screen line. You must not press the RETURN key because this would indicate that the end of the PRINT statement had been reached. A single numbered line in a BASIC program can occupy up to 240 characters (6 screen lines in MODE 7).

The above PRINT statement contains a list of three items to be printed – two strings and an expression. The items are separated from each other by semicolons. When the PRINT statement is obeyed, each item in the list tells the computer to print something. A string between double quotation marks is copied onto the screen character by character exactly as it appears in the program. If an item is not enclosed in quotation marks, then the computer prints a value, possibly after doing a calculation to obtain the value. An item in a PRINT list may be just a single variable name, in which case the value stored in the variable is printed.

When the above PRINT statement is obeyed, the display produced is:

    Area: 20 square metres.

The following table shows what output is produced by various PRINT statements if the variables 'x' and 'y' contain the values shown:

x [7]   y [2]

statement	computer prints
PRINT x	7
PRINT y	2
PRINT "x contains "; x	x contains 7
PRINT "y contains "; y	y contains 2
PRINT "sum of x,y is "; x + y	sum of x,y is 9
PRINT "sum of "; x; ","; y;" is ";x+y	sum of 7,2 is 9
PRINT "x + y = "; x + y	x + y = 9
PRINT x; " + "; y; " = "; x + y	7 + 2 = 9

You should examine carefully the differences in the effects of the last four statements.

Each PRINT statement starts printing on a new line on the screen, unless the previous PRINT statement was terminated by a semicolon. We can also use a PRINT statement with no items. This has the effect of displaying a blank line on the screen (unless the previous PRINT statement terminated with a semicolon in which case it simply starts a new line).

Here is a complete program that calculates a taxpayer's annual tax bill and displays the details on the screen. The input to the program consists of his annual income and his total tax allowance. We assume that tax is deducted at a rate of 30% of taxable income.

```
10     INPUT "Annual income", income
20     INPUT "Tax allowance", allowance
30     CLS
40     PRINT
50     PRINT
60     PRINT
70     PRINT "*****************************"
80     PRINT "Income : "; income
90     PRINT "Allowance : "; allowance
100    PRINT "Taxable income : "; income - allowance
110    PRINT "Tax due : "; (income-allowance)*0.3
120    PRINT "*****************************"
```

The brackets at line 110 are needed to ensure that the allowance is subtracted from the income before multiplication by 0.3. The CLS statement at line 30 has the effect of clearing the screen and the three PRINT statements at lines 40, 50 and 60 insert three blank lines at the top of the screen. Actually, several BASIC statements can be typed on one numbered line of a program, provided that we put colons between the statements. Thus, lines 30 to 60 could be replaced by:

```
30    CLS : PRINT : PRINT : PRINT
```

If you run the above program, you will find that after the initial input has been typed, the computer will produce a display such as:

```
****************************
Income : 9563
Allowance : 2500
Taxable income : 7063
Tax due : 2118.9
****************************
```

It would be nice if numbers representing sums of money were displayed with two digits after the decimal point, but we shall not worry about how to do this for the time being.

If the items in a PRINT statement are separated by commas, then the items printed are more widely spaced than they are if semicolons are used. This facility is needed if we want to tabulate values in columns and we shall leave the details until later. For the time being, you could try replacing the semicolons in the above program with commas and observe the effect. Details about the layout of output produced by PRINT statements appear in Appendix 6.

1.4 Storing the results of calculations

You have seen several programs in which we asked the computer to do simple calculations and print the answers obtained. We tell the computer to do a calculation by writing an arithmetic expression such as:

```
length*width
x+y
(income-allowance)*0.3
```

It is often convenient to arrange for a program to remember the results of a calculation by storing the result in a variable for use later by the same program. For example, a program to add up four examination marks and display the total and the average could be written as:

```
10    INPUT "Four marks:", m1,m2,m3,m4
20    PRINT "Total:"; m1 + m2 + m3 + m4
30    PRINT "Average:"; (m1 + m2 + m3 + m4)/4
```

The symbol '/' is the sign we use for 'divided by'. In this program, the computer adds together the four marks at line 20 and then repeats the same calculation at line 30 before dividing the total by 4. An alternative version of the program is:

```
10      INPUT "Four marks:", m1,m2,m3,m4
20      totalmark = m1 + m2 + m3 + m4
30      PRINT "Total:"; totalmark
40      PRINT "Average:"; totalmark/4
```

The statement at line 20 is known as an 'assignment' statement. The value of the expression on the right of the equals sign is 'assigned to' or stored in the variable whose name appears on the left. Thus the four marks are added together once only and the total is stored in the variable 'totalmark'. The value of this variable is then used by the two PRINT statements and the calculation does not have to be done again.

An important advantage of using variables for storing the values of expressions is that we can give meaningful names to the variables and thus make the meaning of the program more obvious.

As another example, lines 100 and 110 of the tax program could be replaced by:

```
100     taxable = income - allowance
105     PRINT "Taxable income : "; taxable
110     PRINT "Tax due : "; taxable*0.3
```

In the next chapter, we shall look in more detail at the use of assignment statements and mathematical expressions, but the information given above should suffice for the time being.

Exercises

1   Write a program that accepts as input a person's current bank balance and the amount of a withdrawal. The program should print out his new balance.

2   A pay rise of 12.5% has been awarded to a firm's employees and it is to be backdated by 7 months. Write a program to which an employee can supply as input his previous annual salary and which will inform him how much additional backdated pay he should receive.

3   Write a program that draws a Christmas tree outline using asterisks and which displays the message 'A Merry Christmas' in the centre of the tree.

4   Write a program which prints your first initial in the form of a letter several lines high, for example:

```
                JJ
                JJ
                JJ
                JJ
        JJ      JJ
        JJJJJJJJ
          JJJJ
```

5 Write a program that inputs two integers and reports their sum and product in the form of two equations.

For example, if the input is:

    4, 7

the display produced should be:

    4 + 7 = 11
    4 * 7 = 28

6 Write a program that inputs the gross price of an item sold and a discount rate (as a percentage). The program should display a sales invoice, for example:

```
***********************
Gross price      56.25
Discount rate     2.50
Discount          1.41
Discount price   54.84
***********************
```

Do not worry at this stage about producing precisely the above layout.

7 Write a program that displays the perimeter, floor area, wall area and volume of a room given its length, width and height. The program should not repeat any calculations unnecessarily.

1.5 String variables

The programs that we have seen so far have all involved the processing of numerical input. Many important computer applications involve processing input that consists of strings of characters. A character may be a letter of the alphabet, a space, a punctuation mark, or any other symbol that can be typed at a keyboard. For example, a data processing application may involve writing programs that handle lists of names and addresses. Another example is a

word processor which is simply a computer that has been programmed to accept input of typed documents.

In BASIC, we can indicate that we want to store strings of characters in a variable by adding a dollar sign, '$', at the end of the variable's name. Such a string variable can be pictured as a box in which we can store a sequence of up to 255 characters (letters, punctuation marks, etc.). A common use of string variables is to store words. Here is an example of a program that requires input of a word instead of a number:

```
10    CLS
20    INPUT "What is your name", name$
30    PRINT "Hello, "; name$; ". I'm pleased to meet you."
```

If you run this program, it clears the screen and displays the question:

    What is your name?

The question mark has been added by the computer to indicate that it expects some input. It is waiting for you to type some characters to be stored in the variable 'name$'. If you type the word "Jim" and press the RETURN key, then the display will be completed as follows:

    What is your name?Jim
    Hello, Jim. I'm pleased to meet you.

Note the way in which punctuation marks and spaces are inserted at appropriate places in the display by including them inside quotation marks in the PRINT statement. Incidentally, if you want to insert a quotation mark in a string, you must type it twice to indicate that it does not mark the end of the string.

After the computer has obeyed the INPUT statement, the variable 'name$' has the word "Jim" stored in it:

    name$ | Jim |

Whatever name is typed as input for the program is stored in the variable and the program can subsequently be made to do anything we wish with that name. For example, we could extend the program as follows:

```
40    PRINT "Bonjour, "; name$; "."
50    PRINT "Ciao, "; name$; "."
```

An example of the display produced when we run this extended version of the program is:

    What is your name?Alan
    Hello, Alan. I'm pleased to meet you.
    Bonjour, Alan.
    Ciao, Alan.

where the user at the keyboard has typed the word "Alan" after the question mark.

A string does not necessarily consist of a single word. The user of the above program could type first names and surnames before pressing RETURN and that would count as one string. For example:

    What is your name?Harry B. Frackentackle
    Hello, Harry B. Frackentackle. I'm pleased to meet you.
    Bonjour, Harry B. Frackentackle.
    Ciao, Harry B. Frackentackle.

An input string can be terminated by the RETURN key or by a comma. For example, if we run the following program, we can input name and age on one line with a comma in between:

```
10    INPUT "Name and age", n$, age
20    PRINT n$; ", you are "; age; " years old."
30    PRINT "Next birthday, you are "; age + 1
```

The display produced might be:

    >RUN
    Name and age? Fred, 23
    Fred, you are 23 years old.
    Next birthday, you are 24

where the information after the question mark has been typed by the user.

Spaces typed before an input string are ignored. If you want an input string to start with a space or to include a comma, then one way of achieving this is to type the string in quotation marks:

```
10    PRINT "What do you need to buy"
20    INPUT shopping$
30    PRINT "Don't forget to buy the "; shopping$; "."
```

Running the program might produce the display:

What do you need to buy
?"butter, eggs, jam"
Don't forget to buy the butter, eggs, jam.

If we did not use the quotation marks when typing the shopping list after the question mark, then the first comma would be taken to mark the end of the string and the rest of the input would be ignored. Another way of dealing with this problem would be to use the INPUT LINE statement which inputs all the characters on a line, including spaces and punctuation.

Exercises

1 Write a program that accepts input of a name and age and displays a birthday greeting.

2 Write a program that accepts input of two singular nouns and two verbs and displays four simple sentences that use these words. For example, given input:

dog, cat, chase, hate

the program should display:

The dog chases the cat.
The dog hates the cat.
The cat chases the dog.
The cat hates the dog.

1.6 Using the TAB function

While a program is running, the flashing 'cursor', in the shape of an underline character, marks the point on the screen at which the next character (typed by the user or printed by the computer) will be displayed. We can move the cursor about the screen by using the TAB function in PRINT and INPUT statements.

In MODE 7, there are 40 character positions on a line and these are numbered from 0 to 39. One way of using the TAB function is to indicate where on the current line the computer should start printing the next item. For example, the one line program:

```
10    PRINT TAB(4); "Fred"; TAB(20); "Joe"
```

will display the word "Fred" starting at position 4 on the current line. The word "Joe" is displayed starting at position 20 on the same line.

Thus when we use TAB followed by one number in brackets this moves the text cursor along the current line to the position indicated by the number in brackets. If the cursor

is already past the specified position on the current line, then it moves to that position on the next line.

TAB can also be used with two numbers in brackets where the second number is used to select a line for the cursor to move to. In mode 7, the screen is divided into 25 lines numbered from 0 (at the top) to 24 (at the bottom). The following program displays a Christmas message in the centre of the screen.

```
10    CLS
20    PRINT TAB(12,9); "Merry  Christmas"
30    PRINT TAB(18,12); "and"
40    PRINT TAB(12,15); "A Happy New Year"
```

In the PRINT statement at line 20, the presence of 'TAB(12,9)' causes the cursor to be moved to character position 12 on line 9 of the screen before the string "Merry Christmas" is printed. The other PRINT statements use TAB in a similar way.

TAB can also be used in an INPUT statement to move the cursor to a new position before the user types the input required by a program. For example, the behaviour of the following program is interesting:

```
10    CLS
20    PRINT TAB(12,10); "Give me a number"
30    INPUT TAB(20,12); number
40    PRINT TAB(16,12); "2 x "; TAB(15,14);
                                "= "; 2*number
```

You should run this program and make sure that you understand why it behaves as it does. The question mark in the display can be removed by omitting the comma in line 30 as follows:

```
30    INPUT TAB(20,12) number
```

In fact lines 20 and 30 could be combined into a single INPUT statement as follows:

```
20    INPUT TAB(12,10), "Give me a number",
             TAB(20,12)  number
```

As a final example of the use of TAB, the following program illustrates how TAB can be used in PRINT statements that display a simple questionnaire on the screen. TAB is then used in INPUT statements to move the cursor around as the user fills in the questionnaire. As it stands, the program does nothing with the information that is input, but

one of the exercises below asks you to extend the program.

```
10    CLS
20    PRINT TAB(6,1); "Please fill in the following"
30    PRINT TAB(13,3); "Questionnaire."
40    PRINT TAB(13,4); "--------------"
50    PRINT TAB(0,8); "Name:";
60    PRINT TAB(31); "Age:"
70    PRINT TAB(0,11); "Address:"
80    PRINT TAB(0,16); "No. of O-levels:";
90    PRINT TAB(20); "No. of A-levels:";

100   INPUT TAB(5,8), name$
110   INPUT TAB(35,8), age
120   INPUT TAB(8,11), addline1$
130   INPUT TAB(8,12), addline2$
140   INPUT TAB(8,13), addline3$
150   INPUT TAB(16,16), olevels
160   INPUT TAB(36,16), alevels
```

Note the semicolon at the end of the PRINT statement on line 50. This makes sure that, when the program is being obeyed, the PRINT statement does not start a new line and 'TAB(31)' is all that is needed for the next PRINT statement to print further along the same line. The same thing has been done at line 80.

Exercises

1 Write a program that inputs a message (for example, "Happy Birthday", "Happy Anniversary", etc.) followed by a name and displays a personalised version of the message in the centre of the screen, for example:

<div align="center">

****

Happy Birthday

Jim

****

</div>

2 Write a program that displays a screenful of instructions explaining how to play your favourite arcade game. Use TAB to obtain a visually pleasing layout.

3 Extend the questionnaire program so that it uses the input information to calculate a new employee's salary, where the salary is made up of a basic monthly rate of pay, an increment for each year of the employee's age

over 16, an increment for each O-level and an increment for each A-level. The program should display a message welcoming the employee to the company and telling him what his monthly salary will be.

4 Write a program that displays a questionnaire about the day's weather and accepts information typed by the user. Information required might include temperatures at different times of day, rainfall, wind speed, a verbal description of the cloud conditions, and so on. (This program could be part of a more complex program that is used for logging weather records on a daily basis.) Your program should terminate by displaying a few sentences summarising the day's weather conditions.

1.7 Elementary graphics

In the introductory chapter, we saw a simple graphics program that produced a line drawing on the screen by using a combination of MOVE and DRAW statements. Here, we provide some further information on the use of such elementary graphics facilities. Advanced graphics facilities are described in detail in Chapter 9, but it is a good idea to get used to using simple graphics (and sound) facilities as soon as possible rather than considering them advanced facilities to be left until later.

Screen coordinates

We saw in the introductory chapter that a point on the screen is described by giving two values.
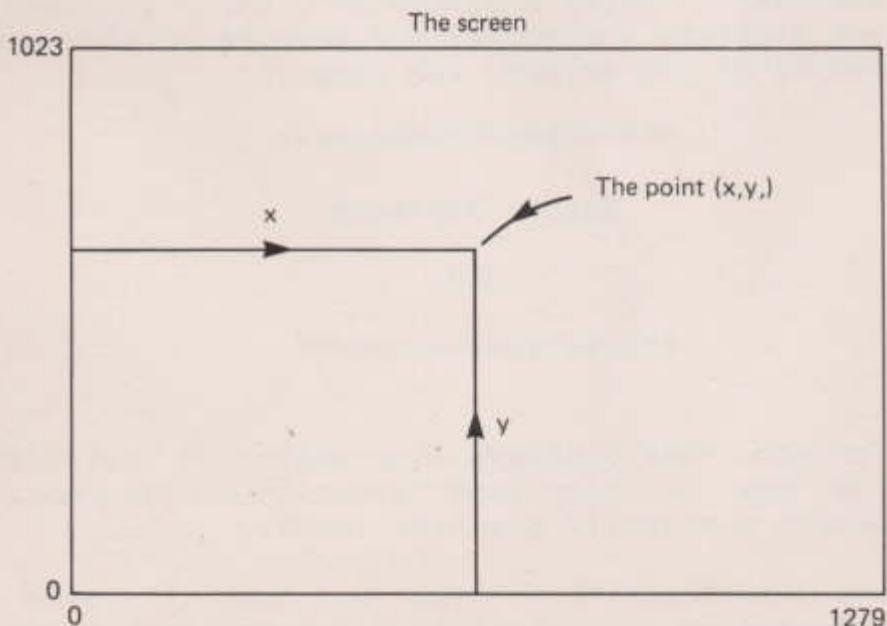
The screen

The first value indicates the horizontal distance of the point from the left of the screen and the second value indicates the height of the point from the bottom of the screen. The first value is usually called the x-value or x-coordinate of the point and the second value is usually called the y-value or y-coordinate of the point. For a point on the screen, its x-value is in the range 0 to 1279 and its y-value is in the range 0 to 1023.

Note that when using graphics, vertical distances are measured from the bottom of the screen, whereas the TAB function measures the vertical position of a character from the top of the screen.

The picture drawn by our introductory program was of fixed size. We can obtain more flexible control over the size and shape of the picture drawn by a graphics program by supplying the dimensions required via an INPUT statement. We have done this in the program below which draws a house. Note that lines 100 and 160 of this program are not instructions for the computer to obey. They are 'remarks' whose purpose is to make the program easier to read and understand.

```
10      PRINT "Dimensions and position of house:"
20      INPUT "Width", width
30      INPUT "Height to eaves", height
40      INPUT "Position: x,y", leftx, bottomy
50      rightx = leftx + width
60      eavesy = bottomy + height
70      roofx = leftx + width/2
80      roofy = eavesy + height/3

90      MODE 5

100       REM Draw outline of walls
110     MOVE leftx, eavesy
120     DRAW leftx, bottomy
130     DRAW rightx, bottomy
140     DRAW rightx, eavesy
150     DRAW leftx, eavesy

160       REM Draw outline of roof
170     DRAW roofx, roofy
180     DRAW rightx, eavesy

190     key = GET : MODE 7
```

We have included blank lines in the program to break it up into sections (see Appendix 1).

The statement

```
key = GET
```

at line 190 is very useful when writing graphics programs. Other uses of GET are covered later, but, in this context, GET makes the computer wait until any key is pressed. The variable 'key' is then given a numeric value that indicates which key has been pressed, although here we do not make any use of this value. Using GET in this way allows the user to look at the picture on the screen and then press any key when he wants the program to switch back to MODE 7.

Drawing accuracy

The edges of the roof drawn by the above program are rather dotted or stepped in appearance. The reasons for such innacuracies are discussed later in Chapter 9. For the time being, you could try changing the MODE statement at line 90 and compare the effects of drawing the picture in different modes. Try

    90   MODE 4

and you should find that the edges of the roof are drawn more accurately. If you have a Model B computer, try using MODEs 2, 1 and 0. MODEs 2 and 5 provide the same drawing accuracy and MODEs 1 and 4 give the same drawing accuracy. MODE 0 provides the greatest drawing accuracy available.

Drawing in different colours

In MODE 5, drawing can take place in one of four different colours. On first entering mode 5, the 'background' colour is black and the 'foreground' colour in which drawing takes place is white. The colours normally available in mode 5 each have a code number as follows:

colour	code number
black	0
red	1
yellow	2
white	3

We can select different colours for drawing by using the GCOL 0 statement. (GCOL stands for Graphics COLour.) For example, if we insert the statement:

    105   GCOL 0,2

in the house drawing program, all the drawing takes place in colour number 2 (yellow). Note that it is the second number in the GCOL statement that selects the colour. For the time being, the first number will always be 0. The significance of this first number will not be discussed until Chapter 9. If we wanted to switch to red before drawing the edges of the roof, we could insert the statement:

165   GCOL 0,1

The walls will now be drawn in yellow and the roof in red.
   In the other graphics mode available on the Model A, MODE
4, only two colours are available - one for background and
one for drawing. These are normally set to black and white.
   On the model B, MODE 0 has only two colours and MODE 1
has the same four colours as MODE 5. MODE 2 has 16 colours,
but we shall not present any detailed information about the
MODE 2 colours at this stage. We leave you to experiment
with them.

Exercises

1 Extend the house drawing program so that it draws a door
   and windows in different colours.

2 The location of the house is specified to our program by
   giving the coordinates of the bottom left hand corner of
   the house. Change the program so that the user specifies
   the position of the centre of the house.

3 Write a program that draws the following car shape:
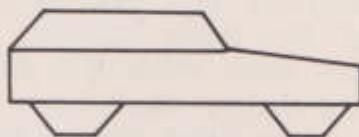
   Use different colours for different parts of the car.

4 Write a program that inputs three numbers and draws the
   outline of a 'bar chart' or 'histogram' where the heights
   of the bars show the sizes of the numbers. For example,
   the bar chart might look like:
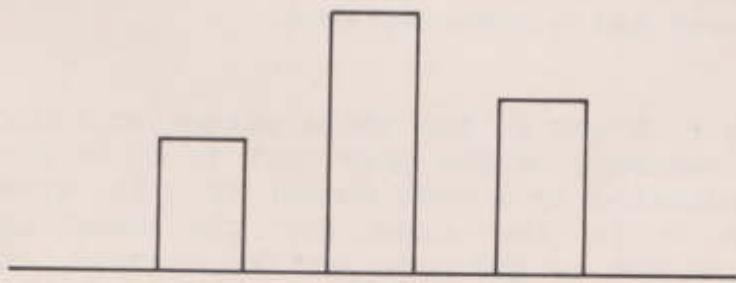
   Use different colours for the three bars.

5 If you are familiar with equations, write a program that
   draws a graph of the equation

        $3x + 2y = 1200$

in MODE 4. Use PRINT and TAB to display the equation alongside its graph. Now write a program that inputs values a, b, c and draws a graph of the equation:

ax + by = c

1.8 Elementary use of sound

If you are not familiar with musical notation, you will have to ignore parts of this section. However, you should still read the rest of it as you may want to use SOUND statements to add sound effects to your programs.

We saw in the introductory section that the SOUND statement is used with four numbers or 'parameters'. The form that a simple sound statement takes can be described as:

SOUND channel, loudness, pitch, duration

Where the significance of each of the four parameters is as follows:

channel
Until we get to Chapter 10, the first parameter will be restricted to one of the four values: 0, 1, 2, 3. The sound generator has four channels and up to four sounds can be made simultaneously, each on a different channel. Channels 1, 2 and 3 are used for producing musical notes while channel 0 can be used for producing different sound effects.

loudness
Until we get to Chapter 10, the second parameter of a sound statement will always be negative or zero. It is used to indicate the loudness of the sound that is to be made. A value of -15 gives maximum loudness, -1 gives the quietest sound and 0 gives silence.

pitch
On channels 1, 2 and 3, the third parameter indicates the pitch or frequency of the note that is to be played. The pitch is indicated by a code number in the range 0 to 255, where 0 is the code for the lowest obtainable frequency and 255 is the code for the highest obtainable frequency. Pitch number 53 corresponds to Middle C on the piano and an increase of 4 in the pitch code corresponds to raising the pitch by one semitone. Thus, for example, the notes in the octave starting at Middle C have the following pitch codes:
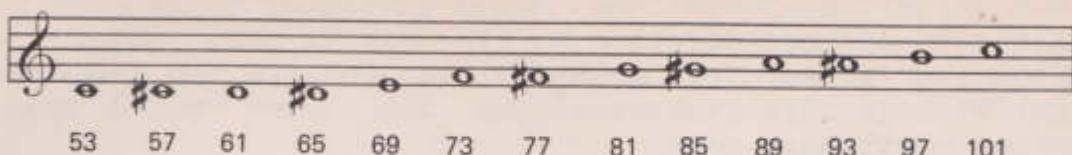
On channel 0, the third parameter selects one of several different sound effects. Only values in the range 0 to 7 are significant.

duration
The last parameter indicates how long the sound should last. Its value can be in the range 0 to 255 where this specifies the duration of the sound in twentieths of a second.

Thus the statement

    10   SOUND 1, -15, 53, 20

causes the note Middle C to be played on channel 1 for 1 second. We have already seen that the following program plays three separate notes of a chord starting on Middle C:

    10   SOUND 1, -15, 53, 20
    20   SOUND 1, -15, 69, 20
    30   SOUND 1, -15, 81, 20

These notes are played consecutively, one after another.
    When the computer obeys a SOUND statement, it immediately tells the sound generator what kind of sound to make. Once it has been told about a sound, the sound generator proceeds independently of the computer and the computer is free to move on to the next statement in the program. We can take advantage of this if we make each of the sound statements in the above program refer to a different channel number:

    10   SOUND 1, -15, 53, 20
    20   SOUND 2, -15, 69, 20
    30   SOUND 3, -15, 81, 20

This program plays the same notes as before, but because they are all on different channels they are played simultaneously (or almost simultaneously - further details in Chapter 10).
    The next program plays the first two bars of a minuet by J. S. Bach: The melody is being played on channel 1 and an accompaniment on channel 2.

The time it takes the sound generator to process the silent note of zero duration at line 110 is just enough to create the effect of a short gap between two identical notes. Without this statement, the two notes would sound like one continuous note.

Playing a longer piece of music in this way would be rather tedious. We shall see better ways of doing this in later chapters.

```
10      SOUND 1, -15, 109, 8
20      SOUND 2, -10, 49, 16
30      SOUND 1, -15, 81, 4
40      SOUND 2, -10, 41, 8
50      SOUND 1, -15, 89, 4
60      SOUND 2, -10, 49, 24
70      SOUND 1, -15, 97, 4
80      SOUND 1, -15, 101, 4
90      SOUND 1, -15, 109, 8
100     SOUND 1, -15, 81, 8
110     SOUND 1, 0, 0, 0
120     SOUND 1, -15, 81, 8
```

On channel 0, the third parameter selects one of several different noises that can be used for creating sound effects. We leave you to experiment with these.

Exercises

1   Experiment with different sound effects on channel 0. Using different combinations of sounds of different durations, try to create effects such as a gunshot and an explosion.

2   Write a program that plays the scale of C major starting on Middle C.

3   Write a program that plays a few bars of any simple tune.

1.9 READ and DATA statements

It is convenient at this stage to mention an alternative way of supplying information to a program, although we shall not make extensive use of this technique until later. You may like to skip this section on a first reading.

Often a program needs to make use of a set of values that are the same each time the program is run, or which change only occasionally. For example, here is a simple program that calculates the tax to be paid given a standard tax-free allowance, a fixed tax-rate and an annual salary.

```
10     INPUT "Allowance", allowance
20     INPUT "Tax rate (percent)", taxrate
30     INPUT "Salary", salary
40     PRINT "Tax due: "; (salary - allowance)*taxrate/100
```

Each time this program is run, the user must type the standard tax-allowance and tax-rate as well as the annual salary. If the first two values are the same each time the program is used, they should not have to be typed each time that the program is run. Such values should be written into the program in some way.

One way of doing this, that will become more useful when large numbers of values are involved, is to use READ statements to obtain values from DATA statements that are included in the program. A READ statement behaves in much the same way as an INPUT statement but, instead of values being typed when the program is run, the values required by READ are listed in a DATA statement and the computer reads them from there. The above program can be written as follows:

```
10     READ allowance, taxrate
20     INPUT "Salary", salary
30     PRINT "Tax due: "; (salary - allowance)*taxrate/100
40     DATA 2150, 32.5
```

The advantage of this may not be apparent at this early stage. After all, we could have used:

```
10     allowance = 2150  :  taxrate = 32.5
```

However, READ and DATA statements are very useful when we write programs that use large lists or tables of values that are the same each time the programs are run. For example, a program that answers enquiries by finding information in a table might include the table in DATA statements. A program that plays a tune might include DATA statements specifying the pitch and duration of each note to be played.

If a program contains several READ statements, each READ starts reading data at the point where the last READ left off. Thus, the following program:

```
10     DATA 5, 7, 9, 4, 13, 19
20     READ a, b
30     PRINT a + b
40     READ c, d
50     PRINT c + d
60     READ e, f
70     PRINT e + f
```

will print:

    12
    13
    32

If a program contains several DATA statements, then data is read from them in turn, in the order in which they appear in the program. Thus the program:

    10   DATA 7, 5
    20   DATA 17
    30   DATA 4, 9, 13
    40   READ a,b,c,d,e,f
    50   PRINT a+b, c+d, e+f

will print

        12   21   22

It does not matter where the DATA statements are inserted in the program. However, it is a good idea to keep DATA statements tidily together at the beginning or end of the program.

Finally the RESTORE statement can be used to tell the program to go back to the start of the DATA. RESTORE followed by a line number tells the program to go back to the start of a particular DATA statement. This facility will be illustrated later when it is needed.

Chapter 2 **Doing calculations**

In this chapter, we describe in further detail the use of arithmetic expressions and assignment statements for doing calculations. We also introduce the functions that are available in BBC BASIC for doing mathematical calculations. If you are not mathematically inclined, then you can safely skip parts of Section 2.4.

2.1 More about assignment statements

We saw in the last chapter that the result of a calculation can be stored in a variable by an assignment statement such as:

```
100    taxable = income - allowance
```

When this statement is obeyed, the expression on the right is evaluated and its value is stored in the variable whose name appears on the left. In the simplest cases, the right-hand side can be a constant or another variable name. For example:

```
sumsofar = 0
```

stores the value 0 in the variable 'sumsofar'.

```
x = y
```

In this example the contents of 'y' are put into 'x' (the previous contents of 'x' being destroyed or overwritten).

Thus, if before the statement is obeyed we have:

x | 2 | y | 3 |
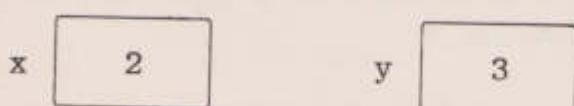
then after the statement is obeyed we have:

x | 3 | y | 3 |

Note that the value in 'y' is left unchanged. We do not 'take out' the value in 'y', but 'copy' it into 'x'.
  Note that a statement such as:

    x = x + y

which may seem somewhat peculiar, is perfectly valid and means: the value of 'x' becomes equal to.what it was before plus the value of 'y'. The right-hand side of an assignment statement is always evaluated first regardless of what variable appears on the left.

Thus if we have:

x ┌──────────┐   y ┌──────────┐
  │   3   │   │   4   │
  └──────────┘   └──────────┘

then immediately after the above statement is obeyed we have:

x ┌──────────┐   y ┌──────────┐
  │   7   │   │   4   │
  └──────────┘   └──────────┘

This idea is used in the following program which illustrates one way of reading and adding two numbers:

```
10    sumsofar = 0
20    INPUT nextnumber
30    sumsofar = sumsofar + nextnumber
40    INPUT nextnumber
50    sumsofar = sumsofar + nextnumber
60    PRINT sumsofar
```

This may seem rather long-winded, but it illustrates a very important point about variables: the value stored in a variable can change while a program is being obeyed. We shall find that the above approach has to be used when large numbers of input values are processed by a program. We shall return to this point in Chapter 4.

2.2 Arithmetic expressions - order of evaluation

In arithmetic expressions used so far, we have sometimes used round brackets. For example, we can write:

    average = (x + y + z)/3

    perimeter = 2*(length + width)

In each case we have used the brackets to clarify our intentions.

We want the computer to calculate:

$$\frac{x + y + z}{3}$$

so we write:

$(x + y + z)/3$

This is simply a consequence of the fact that we are using a keyboard and arithmetic expressions must be typed as a sequence of characters one after another.

If we missed out the brackets:

average = x + y + z/3

the computer would calculate:

$$x + y + \frac{z}{3}$$

which is not what we intended.

In the other example, if we removed the brackets:

perimeter : = 2*length + width

the computer would calculate:

$(2*length) + width$

You can see from this that the computer has rules for dealing with the evaluation of arithmetic expressions.

Let us begin by listing the operators that have been informally introduced so far, together with one new one:

    ^   exponentiation
    *   multiplication
    /   division
    +   addition
    -       subtraction

The exponentiation operator, '^', is used for raising one number to a power. For example,

y = x^3

has the same effect as:

y = x*x*x

The computer can perform only one of the above operations at a time. To perform an operation it requires two operands, which are the quantities on either side of the operator. In the absence of brackets, exponentiation is carried out before multiplication and division, which are carried out before addition and subtraction.

Consider the expression:

    a/b^c + d/e*f

The computer evaluates this as follows:
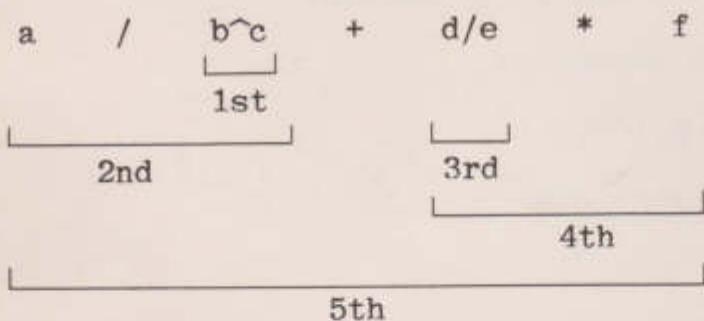
1st result evaluated	b^c
2nd result evaluated	a / 1st result
3rd result evaluated	d/e
4th result evaluated	3rd result * f
5th result evaluated	2nd result + 4th result

Thus the computer evaluates:

$$\frac{a}{b^c} + \frac{d}{e}*f$$

If instead we wanted the computer to evaluate

$$\frac{a}{b^c} + \frac{d}{e*f}$$

then we would use brackets as follows:

    a/b^c + d/(e*f)

Anything inside brackets is evaluated first. For the operators introduced so far, we can summarise this order of 'precedence', as it is called, as follows:

order of precedence: 1st   anything inside brackets
                              2nd   exponentiation
                              3rd   multiplication and division
                              4th   addition and subtraction

Adjacent operators of the same precedence are applied from left to right.

'Adding' strings

At this point it is appropriate to mention that the operator '+' can be applied to two strings. It adds or 'concatenates' the two strings together to produce one longer string. For example,

```
10    INPUT "first name", firstname$
20    INPUT "surname", surname$
30    fullname$ = firstname$ + " " + surname$
40    PRINT "Your full name is", fullname$
```

2.3 Special integer operators

There are two special arithmetic operators that operate only on integers or whole numbers:

DIV   has the same effect as normal division except that any remainder is removed, thus producing an integer result.

MOD   supplies the remainder after dividing two integers

DIV and MOD have the same precedence as '*' and '/'. Consider the following examples:

expression	value
16/5	3.2
16 DIV 5	3
16 MOD 5	1
19/5	3.8
19 DIV 5	3
19 MOD 5	4
8 DIV 3 * 3	6
7 + 5 DIV 3	8
13 - 5 MOD 3	11

The use of these operators on negative numbers is rarely useful. If they are applied to real numbers with fractional parts, the fractional parts are removed before the operators are applied. The use of MOD and DIV on anything other than

positive whole numbers is best avoided.

The following program is a supermarket 'checkout' program that prints the number and denomination of coins required to make up a given amount of change (a whole number of pence). The program indicates the number of 50p, 20p, 10p, 5p, 2p, and 1p coins to be paid out.

```
10      INPUT "Change", change
15
20      noof50s = change DIV 50 : change = change MOD 50
30      noof20s = change DIV 20 : change = change MOD 20
40      noof10s = change DIV 10 : change = change MOD 10
50      noof5s  = change DIV 5  : change = change MOD 5
60      noof2s  = change DIV 2  : change = change MOD 2
70      noof1s  = change
75
80      PRINT "change due is:  no of 50s    "; noof50s
90      PRINT "               no of 20s    "; noof20s
100     PRINT "               no of 10s    "; noof10s
110     PRINT "               no of  5s    "; noof5s
120     PRINT "               no of  2s    "; noof2s
130     PRINT "               no of  1s    "; noof1s
```

We have now introduced seven arithmetic operators and two of these can be applied only to integer quantities. A complete list of operators is given in Appendix 3.

2.4 Standard mathematical functions

A number of predefined mathematical operations or functions are available for use in expressions. The programs for evaluating these standard functions are incorporated as part of your BASIC system.

For example

```
10    x = 4
20    PRINT  SQR(x)
```

prints the value 2.

```
10    y = 5.66
20    PRINT INT(y)
```

prints the value 5.

SQR is the name of the standard function which performs the operation 'finding the square root of'. INT is the name of the standard function which finds the 'integer part' of a number, i.e. it removes the fractional part from a real number to give an integer.

When you refer to a function you use its name and enclose

within round brackets the value to which the function is to be applied. This value, which is called a parameter, can be any arithmetic expression. For example,

```
x = SQR(16)
    .
x = SQR(y)
    .
x = SQR(3*y/z)
    .
x = 15.6 + SQR(3*y/z)
    .
```

Because parameters can be arithmetic expressions, a function can be used in the expression which is the parameter of another function:

```
PRINT INT(SQR(17.3))
```

prints 4.

```
PRINT SQR(SQR(16))
```

prints 2.

```
theta = PI/3
PRINT SQR(SIN(theta) + COS(theta))
```

prints 1.16877089.

You should note that while people usually work in degrees, the parameters of trigonometric functions in BASIC have to be given in radians. The 'predefined constant' PI used above has a value of 3.14159265. A function RAD is available for converting degrees into radians, and its use is illustrated in the next program. For a standard 'inverted v' shaped roof, this program works out the area of roof covering required, given the angular pitch of the roof and the length and width of the building.

```
10      INPUT "Pitch of roof(degrees)", pitch
20      INPUT "Length and width", length, width
25
30      pitch = RAD(pitch)
40      areaofroof = width/COS(pitch) * length
45
50      PRINT "Area of roof covering required is ";
            areaofroof; " sq.m."
```

The following program draws a triangle given the length of the base and the two base angles (in degrees).

```
10      INPUT "Base", base
20      INPUT "Base angles", a1, a2
30      tana1 = TAN(RAD(a1)) : tana2 = TAN(RAD(a2))
40      height = base/(1/tana1 + 1/tana2)
50      topx = height/tana1

60      MODE 4
70      DRAW topx,height
80      DRAW base,0
90      DRAW 0,0
100     key = GET  :  MODE 7
```

Note that without a MOVE statement, drawing starts from the bottom left hand corner of the screen, the point (0,0).

The function INT mentioned above converts a real number into the whole number that is 'less than or equal' to it. If we want to convert a real number, x say, into the 'nearest' integer, we can use the expression:

INT(x+0.5)

This trick is used in the following program which converts degrees Centigrade to degrees Fahrenheit.

```
10      INPUT "Temperature(centigrade)", cdegrees
20      fdegrees = cdegrees * 9/5 + 32
30      PRINT cdegrees; "C = "; fdegrees; "F or approx ";
                                   INT(fdegrees + 0.5); "F"
```

This will produce a display such as:

```
Temperature(centigrade)?21.5
21.5C = 70.7F or approx 71F
```

The following program demonstrates how the function INT can be used to convert a time expressed as a real number of hours into hours and minutes (to the nearest minute). Given a distance between two towns, a speed in miles per hour and a departure time from one town, the program prints the arrival time at the other town. Times are input as, for example, 0845 or 1357 and are output as, for example, 8-45 and 13-57. We assume that the journey takes place in one day.

```
10      INPUT "Distance",distance, "mph",mph,
                          "Start time",startime
20      hours     = startime DIV 100
30      mins      = startime MOD 100
40      realhours= hours + mins/60
```

```
50          REM next statement calculates time of arrival
60          REM as a real number of hours after midnight

70     realhours = realhours + distance/mph

80          REM now convert time back into hours and mins

90     hours     = INT(realhours)
100    mins      = INT((realhours - hours)*60 + 0.5)
110    PRINT "Arrival time: "; hours; "-"; mins
```

There is a complete list of standard functions in Appendix 9.

Exercises

1   A pay rise of 9.9 per cent has been awarded to a company's employees. Write a program to which an employee can supply as input his previous annual salary and which will inform him of his new annual, monthly and weekly rates of pay. (Assume that there are exactly 52 weeks in the year.)

2   Write a program that inputs a length in inches and prints the corresponding length in feet and inches, and then in metres and centimetres (to the nearest centimetre). (1foot = 0.3048 metres)

3   A retiring employee is entitled to an annual pension of one fiftieth of his current annual salary for each complete year's service with the company. All employees start work on the first day of a month and retire on the last day of a month. Write a program into which an employee can type his current annual salary, the month and year he started work with the company and the month and year he retired. A month can be supplied as an integer in the range 1 to 12 The program should inform him what his annual pension will be. HINT: Calculate the total number of months worked and then use DIV to calculate the number of complete years.

4   Write a program that accepts as input the amount of cash (a real number of pounds) to be enclosed in an employee's pay packet. The program should do a 'coin and note analysis' and print the number of coins and notes, of each available denomination, that are to be included in the pay packet.

5   Write a program that calculates the height of a tree, given the distance of a point from the foot of the tree and the angle the top of the tree makes with the

horizontal when viewed from this point.

6 Extend the previous program so that it draws a diagram illustrating the situation.

7 Given a departure and arrival time based on the 24 hour clock write a program that will print out the duration of the journey (assumed to take place all in one day).

2.5 Inventing random numbers: the function RND

Many computer programs are improved if a random element is incorporated in their behaviour. For example, a game playing program should exhibit some degree of unpredictability in its behaviour. A computer assisted learning program should present the user with unpredictable questions. Some interesting graphics effects can be created by programs that incorporate some randomness in their behaviour.

To make the computer invent random numbers, we use the function RND. There are two ways in which we shall use this function. The first method is used for inventing random integers. If n is an integer greater than 1, then RND(n) has a value that is a random integer in the range 1 to n. Thus the following program displays 3 random multiplication questions where each number involved is in the range 1 to 12:

```
10   PRINT RND(12); " x "; RND(12); " ="
20   PRINT RND(12); " x "; RND(12); " ="
30   PRINT RND(12); " x "; RND(12); " ="
```

We shall see in a later chapter how to input answers and test them to see if they are correct. It would be sensible to stop the program printing questions involving 1, such as:

```
1 x 4 =
6 x 1 =
1 x 1 =
```

To do this we need to generate random numbers in the range 2 to 12. This can be done as follows:

```
10   PRINT RND(11) + 1; " x "; RND(11) + 1; " ="
```

Inventing a number in the range 1 to 11 and adding 1 to it gives a number in the range 2 to 12. A similar trick is used in the following program which displays 5 random lines on the screen, each line being in a random colour. To generate a number in the range 0 to 1279, we subtract 1 from a number in the range 1 to 1280.

```
10      MODE 5
20      GCOL 0, RND(3)
30      DRAW RND(1280)-1, RND(1024)-1
40      GCOL 0, RND(3)
50      DRAW RND(1280)-1, RND(1024)-1
60      GCOL 0, RND(3)
70      DRAW RND(1280)-1, RND(1024)-1
80      GCOL 0, RND(3)
90      DRAW RND(1280)-1, RND(1024)-1
100     GCOL 0, RND(3)
110     DRAW RND(1280)-1, RND(1024)-1
```

The other type of random number that we shall
occasionally use is a random real number in the range 0 to
0.9999999. Such a number is produced by using RND with a
parameter 1. For example:

```
10      PRINT RND(1)
```

prints a different real number in this range each time it is
run.

Exercises

1 Write a program that displays a list of 5 random addition
   questions where the numbers involved all lie in the range
   1 to 100. Change the program so that the numbers produced
   all lie in the range 10 to 100.

2 Write a program that plays a sequence of 3 notes, each of
   random pitch and random duration.

Chapter 3 **Choosing alternatives**

Up to now we have seen that a computer program is a list of
statements obeyed one after the other. In this chapter we
will be looking at how we tell the computer to select
particular statements for execution and to ignore others. We
build into the program various alternative courses of action
or pathways and the computer steers a particular course
through the program depending on the outcome of tests that
it makes as it proceeds.

Most practical programs have such facilities built into
them somewhere. This is another way in which programs
achieve generality and flexibility. A program issuing car
insurance for example would have various alternative courses
of action which would depend upon the applicant's age, any
previous claims or convictions and class of car. A program
controlling an industrial process may issue different
control signals depending on input data. 'Increase the gas
supply to a burner if the temperature in an oven is low,
decrease it if it is high' is a simple and common example.

3.1 IF-THEN statements

The simplest BASIC statement involving selection of
alternative courses of action is do-don't selection, i.e.
the computer executes or ignores a statement depending on
the outcome of a test. Consider the following simple
examples:

IF prevcon > 3 THEN fine = fine*2

(The value of 'fine' is doubled if the value of 'prevcon'
is greater than 3 otherwise it is left unchanged)

IF total > 100 THEN total = total - 0.10*total

(The value of 'total' is reduced by 10 percent providing
it is greater than 100 - a discount facility)

IF weight > 200 THEN PRINT "try slimming"

(A message is either printed or not)

40

In each of these examples the form is:

IF [condition] THEN [statement]
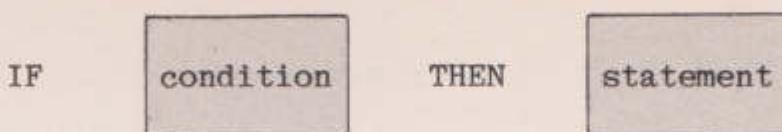
and the behaviour of such simple IF-THEN statements can be illustrated as follows:

test the condition

TRUE   FALSE

obey statement   do nothing

continue with rest of program
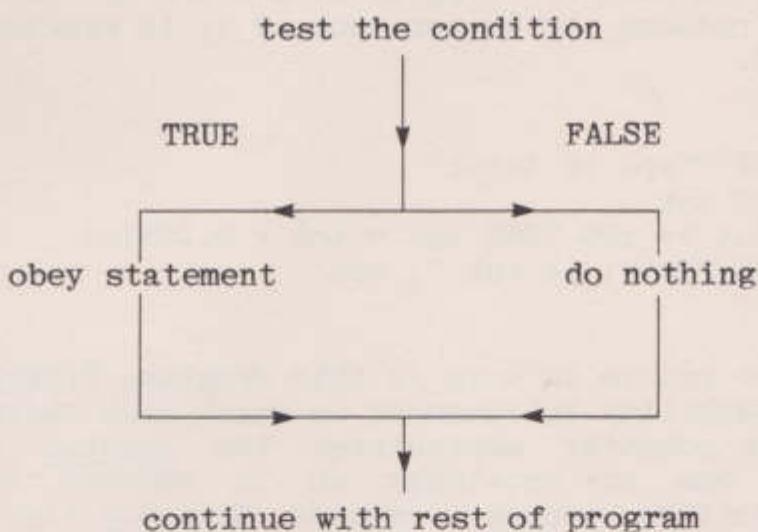
The branch taken depends on the outcome of the test. If the condition is satisfied then the left hand branch is followed, otherwise the right hand branch is taken. We say that the outcome of a test is the value TRUE or the value FALSE and this is a concept of which we shall make frequent use later. A simple condition relates two quantities using one of a number of relational operators. The particular relational operator we used above was '>' which means 'greater than'.
The complete list of relational operators is:

operator	meaning
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
=	equal to
<>	not equal to

The operators >=, <= and <> are single operators written as two characters because of the limited number of characters usually available on a keyboard.
The following examples use these operators:

```
IF age >= 18 THEN PRINT "elig. for jury service"
```

```
IF x <> y THEN PRINT "x and y are unequal"
```

The above are all examples of do-don't selection, the computer executes a statement or does nothing depending on the outcome of a test.

The input to the next program is a price or total. This is to be reduced by 10 per cent if it is greater than or equal to 100.

```
10    PRINT "type in total"
20    INPUT tot
30    IF tot >= 100 THEN tot = tot - 0.10*tot
40    PRINT "Total is now "; tot
```

There are two points to note in this program. Firstly it is your responsibility to remember in which cash units you are working - the computer manipulates the numeric value of 'tot'. It has no knowledge as to whether the value represents dollars, pounds or zlotys. Secondly line 40 will display either the value of 'tot' as input, or an adjusted value.

Here is a program that makes up two random integers in the range 1 to 100 and asks the person seated at the keyboard to add them together. If the wrong answer is typed, the program 'blows a raspberry' (not a very good one).

```
10    a = RND(100) : b = RND(100)
20    PRINT a; "+"; b; "=";
30    INPUT answer
35
40    IF a + b <> answer THEN  SOUND 0,-15,5,50
```

Exercises

1 Write a program that takes as input two amounts representing the current balance of a bank account and a withdrawal. The program is to display the new balance together with a warning if the account 'goes into the red'.

2 Experiment with the SOUND statement used at the end of the last section so as to vary the noise produced.

3.2 IF-THEN-ELSE statements

Now consider the following examples:

    IF age >= 18 THEN PRINT "elig " ELSE PRINT "underage"

    IF age > 20 THEN bonus = pay*0.2 ELSE bonus = pay*0.1

These examples illustrate 'do one or the other' selection. The computer executes the statement following the THEN if the test is TRUE, otherwise (or ELSE) it executes the other. Incidentally, especially when using mnemonic names, an IF-THEN-ELSE statement may not fit on one screen line. However the program line may be broken over several lines on the screen as described in Section 1.3.

    40   IF answer = a + b THEN
                PRINT "That's right!"
            ELSE SOUND 0,-15,5,50

The whole statement must be typed without pressing the RETURN key, extra spaces being used to take us onto the next line where necessary. Remember that a BASIC statement can not occupy more than 6 screen lines in MODE 7.

The form for a simple IF-THEN-ELSE statement is:

IF | condition | THEN | statement T | ELSE | statement F
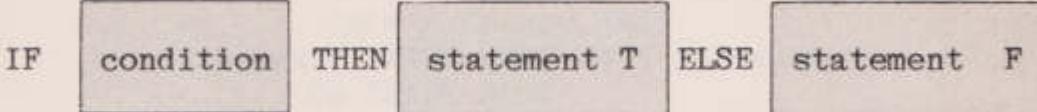
Its behaviour can be illustrated as follows:
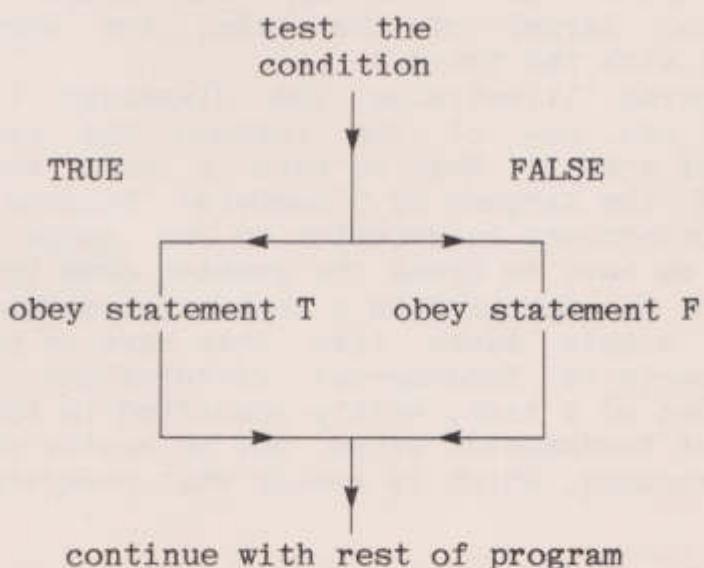
Again we are selecting one out of two alternatives but this time the second alternative, instead of being to do nothing,

is another statement. Again the selection is dependent on the outcome of the test which can be either TRUE or FALSE. 'Statement T' is executed if (the value of) the condition is TRUE, otherwise 'statement F' is executed because (the value of) the condition is FALSE. Here are some programs illustrating these facilities.

This discount program is like the previous except that if the total is less than 100 it is reduced by 5 per cent instead of being left unaltered.

```
10    PRINT "type in total"
20    INPUT tot
30    IF tot >= 100 THEN tot = tot - 0.10*tot
                    ELSE tot = tot - 0.05*tot
40    PRINT "After discount, total is "; tot
```

The next program uses two IF-THEN-ELSE statements. It takes as input 3 numbers and displays the largest of these three numbers.

```
10    PRINT "Type in 3 numbers"
20    INPUT n1, n2, n3
30    IF n1 > n2 THEN maxsofar = n1
                 ELSE maxsofar = n2
40    IF maxsofar > n3 THEN PRINT maxsofar
                       ELSE PRINT n3
```

The first IF-THEN-ELSE statement puts the larger of the first two numbers into 'maxsofar'. The second IF-THEN-ELSE compares the larger of the first two numbers (now in 'maxsofar') with the third.

This program illustrates the classical limitation of computers and one of the reasons why programming is something of a skill. Here we have a very simple logical task 'find the largest of 3 numbers'. Because the computer can only perform one calculation or one comparison at any one time we have to break the problem down into 2 pairwise comparisons. The computer is a sequential machine and even apparently simple tasks like this have to be broken down into a sequence of fundamental calculations. It is this breaking down of a task, easily specified in English, into a sequence of fundamental steps, not so easily specified in a computer language, which is really what programming is all about.

Exercises

1 Write a program that displays an addition question and makes one noise if an answer typed at the keyboard is

correct and a different noise if the answer is wrong. Experiment with the SOUND statement until you find two appropriate noises.

2 Write a program to print an electricity bill given a present and previous meter reading. The program should take account of the fact that a meter goes back to zero after reaching 99999, making the previous reading greater than the present.

3 Write a program to accept two items of information: a British shoe size and a 1 or 0 to indicate a man's or woman's shoe. The program is to output the American size according to the table:

```
         Men's shoes
British      7    8    9   10    11
American   7.5  8.5  9.5 10.5  11.5     (ie. add 0.5)

         Women's shoes
British      3    4    5    6     7
American   4.5  5.5  6.5  7.5   8.5     (ie. add 1.5)
```

4 Write a program that accepts four examination marks and prints out a 'pass/fail' message depending on whether the average is greater than or equal to 50, or less than 50.

3.3 Comparing strings

The next program is identical to the previous one except that the numeric variables have been changed to string variables. The input to the program is three names. The program carries out exactly the same operations as the previous one, but on strings instead of numbers.

```
10    PRINT "Type in 3 names"
20    INPUT n1$, n2$, n3$

30    IF n1$ > n2$ THEN maxsofar$ = n1$
                   ELSE maxsofar$ = n2$

40    IF maxsofar$ > n3$ THEN PRINT maxsofar$
                         ELSE PRINT n3$
```

If you execute this program and type in 3 names, in capital letters, you will find that it prints the name that comes last alphabetically. When comparing strings, the operator '>' is best read as 'comes after alphabetically'.
Computers usually store letters using a code that increases in unit steps. In the case of capital letters the

lowest code represents "A" and the highest code represents "Z". Therefore, as far as computers are concerned, the 'highest' capital letter in the alphabet is "Z" and the 'lowest' is "A". When the computer is comparing names it compares on the basis of the first character in each name. Thus, "JOHN" comes after "ALAN" because "J" is further on in the alphabet than "A". If the first character in each name is the same, then it compares on the basis of the second characters, and if they are the same, then it compares on the basis of the third etc. Thus "HEART" comes after "HEARS".

A similar ordering applies to the lower case letters. Thus "a" comes before "b" which comes before "c" and so on. However, all the capital letters come before the lower case letters, so that "A", "B", ..., "Z" all come before "a". This means that you must be careful if you want to compare words, some of which start with a capital letter and some of which start with a small letter. All the relational operators that are used to compare numeric entities can be used in the context of strings. You should be able to work out their effect on strings.

The following program tests two strings for equality. It tests whether someone seated at the keyboard knows who wrote 'War and Peace'.

```
10    PRINT "Who wrote WAR AND PEACE";
20    INPUT answer$
30    IF answer$ = "Tolstoy" THEN PRINT "Correct!"
                             ELSE PRINT "Rubbish!"
```

The answer has to be typed exactly as the program expects it. "L. Tolstoy" or "Leo Tolstoy" will not be accepted. More advanced string comparison facilities will be discussed later in Chapter 8.

Manipulations on strings are often used in commercial programming: for example, sorting a list of names into alphabetical order for insertion in a telephone directory or sorting a list of names and addresses according to cities.

Exercises

1 Write a program to process a banking transaction. The input is to consist of 3 items of information: a balance, an amount, and, the character "c" or "d" depending on whether the amount is a credit or a debit. The program is to print out the new balance.

2 Change the first program of the last section so that it selects and prints the name that comes first alphabetically. Now extend the program so that it accepts input of four names and prints the one that comes first

alphabetically.

3.4 Compound statements

So far we have looked at 'do-do nothing' conditional statements and 'do one or the other' conditional statements. In each case the thing that was obeyed or ignored was a single statement. Suppose we have a context where more than one statement is to be obeyed conditionally. For example what if we wanted to sort a set of 3 names into alphabetical order? We could do this as follows: Say we have the 3 names in 'n1$', 'n2$' and 'n3$' and we want to end up with the 'first alphabetically' in 'n1$' and the 'last alphabetically' in 'n3$'. Remember that in computer programs we can only perform manipulations like comparisons between two entities one at a time. To get the names into alphabetic order we compare 'n1$' and 'n2$', and if they are in the wrong order, swap their contents over. We perform the same operation with 'n2$' and 'n3$', and then with 'n1$' and 'n2$' again. This will guarantee that the names end up in alphabetic order. Try it on a bit of paper.
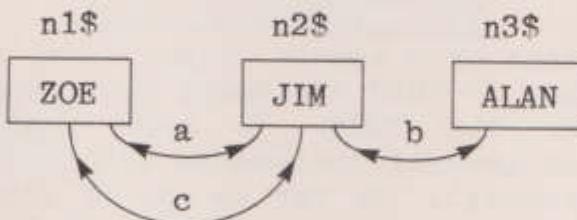
a	:	1st conditional swap
b	:	2nd " "
c	:	3rd " "

We could summarise the program as:

```
IF  n1$  > n2$  THEN swap them
IF  n2$  > n3$  THEN swap them
IF  n1$  > n2$  THEN swap them
```

Now the point about this example is that we need three BASIC statements to swap the contents of 2 variables:

```
temp$  = n1$
n1$    = n2$
n2$    = temp$
```

swaps the contents of n1$ and n2$. You may have thought that
two statements such as:

```
n1$  = n2$
n2$  = n1$
```

would suffice, but if you try this out on a piece of paper
you will find that you end up with identical contents in
each variable! So given that we now have 3 statements to be
conditionally obeyed or associated with each IF statement we
need a way of tying these 3 statements together - making
them behave as a single statement. We do this by using
colons:

```
10    IF n1$ > n2$ THEN temp$ = n1$:
                       n1$ = n2$:
                       n2$ = temp$
```

The effect of using colons is to make the three statements
behave as one:

```
10    IF n1$ > n2$ THEN    obey 3 statements
                          as if they  were
                          a single statement
```

Note that the restriction of 6 screen lines per statement
applies to the complete IF-THEN statement including all the
statements listed after the THEN. In more sophisticated
languages a grouping of several statements into one is known
as a 'compound statement'. As far as the IF statement is
concerned the three statements are treated as if they were a
single 'compound statement'.
   Here is a complete program for sorting three names into
alphabetic order.

```
10    PRINT "Type in 3 names"
20    INPUT n1$, n2$, n3$

30    IF n1$ > n2$ THEN   temp$ = n1$:
                         n1$ = n2$:
                         n2$ = temp$

40    IF n2$ > n3$ THEN   temp$ = n2$:
                         n2$ = n3$:
                         n3$ = temp$

50    IF n1$ > n2$ THEN   temp$ = n1$:
                         n1$ = n2$:
                         n2$ = temp$

60    PRINT "Sorted names are" ' n1$, n2$, n3$
```

This program serves as an illustration of compound statements; realistic techniques for sorting large numbers of names into order are mentioned in Chapter 6. Of course you need not use as extravagant a program layout as this; we have used such a layout to make clear the meaning of the program.

The general form of an IF-THEN with a compound statement is thus:

IF [condition] THEN [any number of statements separated by colons (and within the line limitation)]
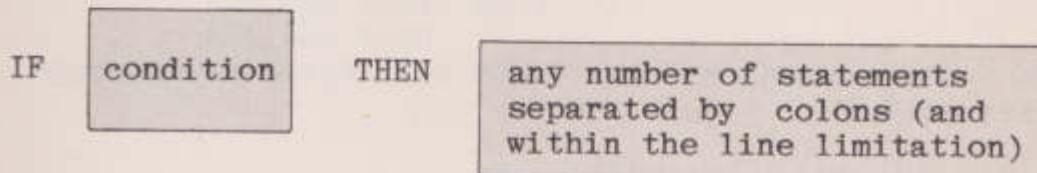
We have already mentioned that, even if we are not using an IF-statement, we can use colons to group several statements together on one line:

```
10    INPUT a, b
20    sum = a + b : diff = a-b : prod = a*b
30    PRINT sum, diff, prod
```

The same facility can be used with an IF-THEN-ELSE statement and the next programs illustrate this.

Here is a program that processes a bank account using different interest rates depending on whether the account is overdrawn or not. Also it prints a message suggesting either an overdrawn letter or a statement.

```
10    INPUT balance
20    IF balance < 0 THEN
          PRINT "Write overdrawn letter" :
          balance = balance + 0.025*balance
      ELSE
          PRINT "Send statement" :
          balance = balance + 0.018*balance
30    PRINT "Balance is now "; balance
```

The next program draws a square or a triangle as specified by someone at the keyboard.

```
10      INPUT "square or triangle", shape$
20      MODE 5
30      MOVE 400,400
40      DRAW 800,400            :REM draws the base line.

50      IF shape$ = "square" THEN    DRAW 800,800 :
                                     DRAW 400,800 :
                                     DRAW 400,400
                             ELSE    DRAW 600,800 :
                                     DRAW 400,400
60      keypressed$ = GET$    :REM wait until key pressed.
70      MODE 7
```

The general form is:

IF | condition | THEN | any number of
                             statements joined
                             with colons |

                    ELSE | any number of
                             statements joined
                             with colons |
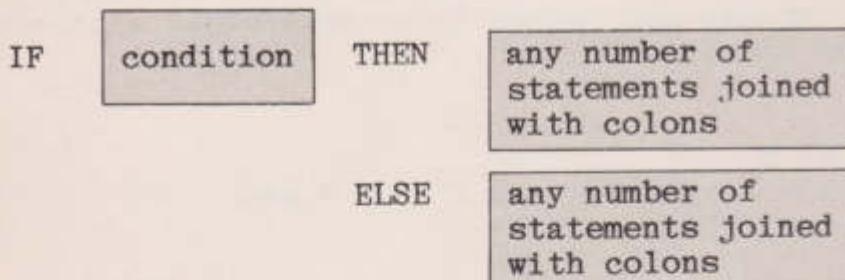
A more widely available BASIC statement used for obtaining
the above effect is the GOTO statement (see later). The
above method is more elegant and results in considerably
more readable programs. Even when forced to use GOTO
statements, it is still helpful to plan a program in terms
of IF-THEN or IF-THEN-ELSE statements.

Exercises

1 Write a program that finds the sum and difference of two
   numbers, first ensuring that the larger of the two
   numbers is in the variable 'larger' and the smaller is in
   the variable 'smaller'.

2 Extend the 'square or triangle' program so that the user
   specifies the colour to be used (red or yellow) in
   drawing the shape.

3.5 More complicated conditions

The simple conditions we have used so far have been composed
of two quantities related by a relational operator:

| quantity1 | relational operator | quantity2 |
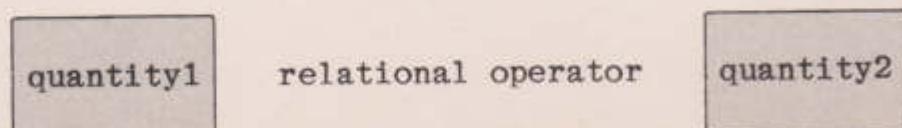
The quantities we have used have been variables but they can also be arithmetic expressions:

```
IF   sum1 + sum2 - credit > 250 THEN
        PRINT "Credit limit exceeded"


IF age - 60 < 5 THEN
    benefit = lumpsum * 1.5
ELSE
    benefit = lumpsum


IF total > 1.5 * currentbalance THEN
    PRINT "Credit limit exeeded."
```

We can combine more than one condition in an IF statement by joining conditions together using the words AND and OR:

```
IF previousconvictions > 3 AND timespread < 1.5 THEN
    fine  = fine * 4


IF weight > 200 AND height < 1.7 THEN
    PRINT "you are overweight"
ELSE
    PRINT "your weight is reasonable"


IF weight > 200 OR dailycalories > 2000 THEN
    PRINT "cut down"
ELSE
    PRINT "ok"


IF x = y AND x > 0 AND y > 0 THEN
    PRINT "x and y are equal and positive"
```

Remember   that   each   complete   IF-THEN   or   IF-THEN-ELSE statement represents a single numbered line of BASIC program even where it occupies several screen lines. The words AND and OR join individual conditions together to make a more complicated condition. Technically, a condition is called a logical expression.

A common mistake made by new programmers is to write:

```
IF x > 0 AND < 10 THEN ...
```

instead of:

```
IF x > 0 AND x < 10 THEN ...
```

You should see from this that each constituent simple condition involving a relational operator must be complete.

The operators AND and OR are universally available in all high level languages. BBC BASIC allows another logical operator EOR, the mnemonic for 'exclusive or'. To appreciate the difference between EOR and OR try running a small program that includes:

```
IF weight > 200 EOR dailycalories > 2000 THEN
    PRINT "cut down"
```

and note the difference in effect between using the EOR and OR operators. The OR operator includes the case when both relational expressions are TRUE, whereas the EOR operator excludes this case. In the context of this example the use of the OR is correct. The use of EOR is clearly erroneous. In practice, you will find that the EOR operator is very rarely needed.

3.6 Things you need to know about AND, OR (and EOR)

When an IF statement contains a condition involving AND and OR, the meaning of the condition is usually clear from reading the program. However, here we tabulate the possible values of a composite condition involving two subsidiary conditions.

```
IF    condition1    AND    condition2    THEN......
```

condition1	condition2	composite condition
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE

```
IF    condition1    OR    condition2    THEN......
```

condition1	condition2	composite condition
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	TRUE

```
IF     condition1     EOR     condition2     THEN......
```

condition1	condition2	composite condition
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	FALSE

Try making up your own examples using combinations of AND, OR and EOR. The words AND, OR and EOR are called logical operators and they join conditions or logical expressions together, just as arithmetic operators join arithmetic expressions together.

The other logical operator we use is NOT. Its use can be illustrated by a simple example:

```
IF NOT (x = y) THEN
   PRINT " x and y are unequal"
```

is exactly equivalent to:

```
IF x <> y THEN
   PRINT " x and y are unequal"
```

As with arithmetic operators, there is an order of priority for logical operators. The order of priority is NOT, AND, OR(or EOR). Thus:

```
IF calories>2000 OR weight>200 AND height<1.7 THEN ...
```

is equivalent to:

```
IF calories>2000 OR (weight>200 AND height<1.7) THEN ...
```

and not to:

```
IF (calories>2000 OR weight>200) AND height<1.7 THEN ...
```

Remember: if in doubt, you can use extra brackets to make your intentions clear. The use of redundant brackets can often help to make a complicated condition more readable.

There is in fact an overall order of precedence defined for the complete set of BBC BASIC operators (see Appendix 3).

Exercises

1 An insurance policy is to be issued only if the applicant is over 18, has a car with an engine size under 2000 and has less than 3 motoring convictions. Write a program that inputs the relevant information and states whether

or not a policy is to be issued.

2   Extend the last example so that if the policy is refused
    a reason is given.

3.7 Logical variables

Up to now we have stored integer and real numbers in
variables. You can also store the values TRUE or FALSE in a
numeric variable. (These values are in fact represented
inside the computer by numbers.) Although this is not a
strictly necessary facility it can be used to make programs
easier to read. The use of such a logical variable is
illustrated in the next fragment which prints out part of a
menu. The dishes on the menu are to vary according to
whether or not it is a summer month.

```
10    INPUT month
20    itsasummermonth = (month >= 5 AND month <= 8)

30    IF itsasummermonth THEN PRINT "Melon"
                         ELSE PRINT "Oysters"

40    PRINT "Roast chicken with ";
50    IF itsasummermonth THEN PRINT "green salad"
                         ELSE PRINT "two veg"
```

When the program is being obeyed, the value of the
condition:

month >= 5 AND month <=8

is TRUE or FALSE. This value (actually -1 or 0) is stored in
the variable 'itsasummermonth'. Apart from the program being
easier to read and develop, it can refer to the value of the
condition as often as is necessary without having to perform
the test again. The brackets in line 20 are not strictly
necessary, but they make the statement easier to read. The
next fragment uses logical variables together with the
operator NOT.

```
10    INPUT height, weight
20    tall = height > 1.8
30    heavy = weight > 200

40    IF tall AND heavy THEN
         PRINT "You are big enough to be a policeman"
50    IF NOT tall AND NOT heavy THEN
         PRINT "Have you thought about being a jockey?"
```

The next program would involve considerable testing if the logical variables were not used.

An insurance broker wishes to implement the following guidance table in a program, so that when he types in an age, engine capacity, and number of convictions the appropriate message is displayed.

age	engine size	convictions	message
>=21	>=2000	>=3	policy loaded by 45%
>=21	>=2000	< 3	policy loaded by 15%
>=21	< 2000	>=3	policy loaded by 30%
>=21	< 2000	< 3	no loading
< 21	>=2000	>=3	no policy to be issued
< 21	>=2000	< 3	policy loaded by 60%
< 21	< 2000	>=3	policy loaded by 50%
< 21	< 2000	< 3	policy loaded by 10%

```
 10 INPUT age, cc, convictions
 20 over21  = age >= 21
 30 largecar = cc >= 2000
 40 riskdriver = convictions >= 3

 50 IF over21       AND      largecar AND      riskdriver
    THEN PRINT "Policy loaded by 45 percent"

 60 IF over21       AND      largecar AND NOT riskdriver
    THEN PRINT "Policy loaded by 15 percent"

 70 IF over21       AND NOT largecar AND      riskdriver
    THEN PRINT "Policy loaded by 30 percent"

 80 IF over21       AND NOT largecar AND NOT riskdriver
    THEN PRINT "No loading"

 90 IF NOT over21 AND      largecar AND      riskdriver
    THEN PRINT "No policy to be issued"

100 IF NOT over21 AND      largecar AND NOT riskdriver
    THEN PRINT "Policy loaded by 60 percent"

110 IF NOT over21 AND NOT largecar AND      riskdriver
    THEN PRINT "Policy loaded by 50 percent"

120 IF NOT over21 AND NOT largecar AND NOT riskdriver
    THEN PRINT "Policy loaded by 10 percent")
```

3.8 The GOTO statement

In this very brief section, we mention what must be one of the most widely used (and abused) statements in the BASIC language - the GOTO statement. This statement is used to tell the computer to transfer its attention to a different line of the program. For example:

```
50      GOTO 120
```

tells the computer to obey the statement at line number 120 next and carry on from there. We shall use this statement only when it is absolutely essential (for example, in the next section). BBC BASIC provides other language constructions that cover the vast majority of programming situations - USE THEM!

Although the GOTO statement looks deceptively simple, excessive use of GOTO statements leads to programs whose possible execution pathways are so intertwined that they are difficult to read, difficult to debug and difficult to modify.

3.9 Selecting one of many alternatives - ON-GOTO

There are many contexts in which we require a statement to select one out of a number of alternatives, rather than one out of two alternatives. One way of doing this can be represented diagrammatically as:

```
                          selector
                             |
                             v
        +------------+------------+------------+
        |            |            |            |
        v            v            v            v
   statement1   statement2   statement3   statement4 ...etc.
        |            |            |            |
        +------------+------------+------------+
                             |
                             v
        continue with rest of the program
```
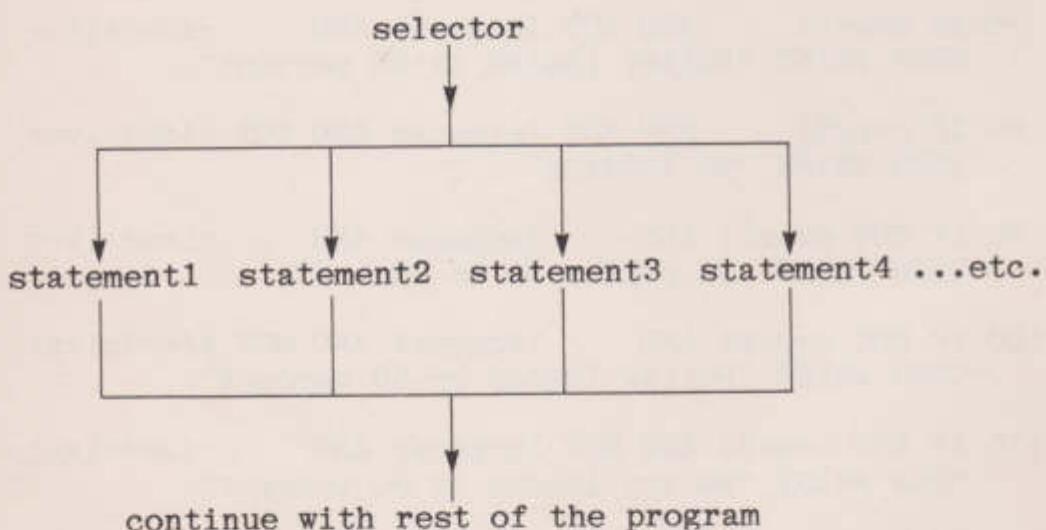
We have replaced a condition which could have one of two values - TRUE or FALSE - with a more general selector. The two-valued condition which selected one out of two branches has been replaced by an entity which can select one out of a number of branches.

Consider the following program fragment that converts a

date typed in as, for example:

    3, 2

into

    3 February is the date.

```
1    INPUT day, month
2    PRINT day;
3    ON month GOTO  10, 20, 30, 40, 50, 60,
                    70, 80, 90, 100, 110, 120

10      PRINT " January"  ; : GOTO 130
20      PRINT " February" ; : GOTO 130
30      PRINT " March"    ; : GOTO 130
40      PRINT " April"    ; : GOTO 130
50      PRINT " May"      ; : GOTO 130
60      PRINT " June"     ; : GOTO 130
70      PRINT " July"     ; : GOTO 130
80      PRINT " August"   ; : GOTO 130
90      PRINT " September"; : GOTO 130
100     PRINT " October"  ; : GOTO 130
110     PRINT " November" ; : GOTO 130
120     PRINT " December" ;

130    PRINT " is the date."
```

The effect of the ON-GOTO statement should be fairly
obvious. Line 3 causes the program to select one out of the
lines 10 to 120 depending on the value in 'month'. We shall
call the variable 'month' the selector. The effect of the
ON-GOTO statement is to cause the program to select the n'th
statement number in the GOTO list, where n is the value of
the selector. What happens if the selector contains a number
that is greater than the number of items in the GOTO list?
In this event an execution error will occur. It is always
good programming practice to prevent execution errors and
make the program print an error message. If a program error
message is printed rather than an operating system message,
then it can always be specific to the program context. This
will always be easier for a user of the program to
understand and interpret. In this case prevention of an
execution error can be achieved by using an ON-GOTO-ELSE
statement. Line 3 would then be:

```
3     ON month GOTO 10, 20, 30, 40, 50, 60,
                    70, 80, 90, 100, 110, 120
      ELSE PRINT " You have typed an erroneous month" :
           PRINT " Please rerun the program" :   END
```

Now the proliferation of 'GOTO 130's in the above example may seem a bit clumsy, but in fact they are not redundant. Without a 'GOTO 130', the program would carry on and obey the statement immediately after the one selected. After a particular statement is selected, a GOTO can transfer control to anywhere in the program, i.e. all the GOTOs could specify different destinations. This is not, however, recommended practice – the ON-GOTO then becomes a sort of complicated junction where the program control goes in through a single entrance and out through one of several different exits. As we have already stated, this sort of thing makes a program very difficult to read, debug or modify.

3.10 Tricks with the ON-GOTO statement

Although the ON-GOTO statement in BASIC is rather restrictive compared with the equivalent facility in more sophisticated languages, it can be enhanced by using various tricks. For example consider the following:

```
1     INPUT month
2     ON month GOTO 10, 10, 20, 20, 20,
                    30, 30, 30, 30,
                    20, 10, 10

10        PRINT "low season rate"    : GOTO 40
20        PRINT "mid season rate"    : GOTO 40
30        PRINT "peak season rate"
40        .
          .
```

This may be part of a holiday booking program. A month is input to the program and the program displays one out of 3 messages. Here we have only 3 statements but the selector can take 1 out of 12 values. The equivalence between the 12 values in the selector and the 3 statements is achieved by duplicating the appropriate line numbers in the ON-GOTO statement.

The next program uses an arithmetic trick to select one out of 4 interest rates.

The yearly rate of interest on a loan is:

```
        0 <   loan <  1000  -  10 %
     1000 <=  loan <  2000  -  11 %
     2000 <=  loan <  3000  -  11.5 %
     3000 <=  loan <  4000  -  11.75 %
     4000 <=  loan <  5000  -  12 %
```

The program works out a year's interest given the size of
the loan. Note that the interest rate could not be
conveniently calculated using a single expression because
the variation of rate with loan size is not linear.

```
10     INPUT loan
20     ON (INT(loan/1000) + 1)  GOTO 21, 22, 23, 24, 25

21        intrate = 10    : GOTO 30
22        intrate = 11    : GOTO 30
23        intrate = 11.5  : GOTO 30
24        intrate = 11.75 : GOTO 30
25        intrate = 12

30     PRINT " Interest to pay "; loan*intrate/100
```

Finally if the value that is to used to select a
statement is not a small integer, and can not be easily
converted into one, a construction like the following might
be useful:

```
5        INPUT m$
10       IF m$="Jan"  THEN
            PRINT "January"   : GOTO 130
20       IF m$="Feb"  THEN
            PRINT "February" : GOTO 130
30       IF m$="Mar"  THEN
            PRINT "March"     : GOTO 130
            .
            .
            .
120      IF m$="Dec"  THEN
            PRINT "December"
130         .
            .
            .
```

The selector is tested against each possible value that it
could take. With this construction, the GOTO statements
could be omitted. If efficiency is a consideration, the use
of the GOTOs stops the program wasting time testing the
selector against other values when a match has been found.
This construction, especially without the GOTOs, is rather
more readable than the others and we shall tend to use it
even where an ON-GOTO would have been appropriate. Program

readability is a feature that we shall be stressing throughout the text.

Exercises

1 Extend the date conversion example above to add the appropriate suffix to the day. For example, given input:

    1, 6

the program should display '1st June'.

2 The following program fragment illustrates how the computer can select a random question and ask it:

```
10    question = RND(3)
20    PRINT "What was the date of the Battle of ";
30    ON question GOTO 31, 32, 33
31      PRINT "Hastings";:correctanswer=1066 : GOTO 40
32      PRINT "Bannockburn";:correctanswer=1314 : GOTO 40
33      PRINT "Gettysburg";:correctanswer=1863
40    .
      .
```

The value of the variable 'correctanswer' can then be used to check an answer typed at the keyboard. Use this idea to write a program that asks a random question about some topic such as Capital Cities, Authors, Painters, Composers, Dates of Events, etc.

Chapter 4 **Loops**

Loops are one of the most commonly used structures in computer programs. A loop is the colloquial name given to a structure which makes a program do things over and over again. One of the original mechanical computers was called a calculating engine - perhaps a more apt term than computer - having as it does the connotation of a rotating machine churning out numbers. Most practical programs repeat some operations over and over again, albeit on different data each time. A payroll program, for example, may perform exactly the same operations for many employees. A program searching for the occurrence of certain words or phrases in a novel performs the same comparisons over and over again until it reaches the end of the novel. So it is natural that we should have loop facilities available for use in a program.

4.1 Deterministic loops (FOR statements)

Suppose we want to write a program which adds three numbers together. We could write:

```
10    INPUT  n1, n2, n3
20    sum = n1 + n2 + n3
30    PRINT "Sum is "; sum
```

However, if we wanted a program to add 100 numbers together this technique would be somewhat clumsy! If we restructure our summing operation for three numbers like this:

```
5     sum = 0

10    INPUT n
20    sum = sum + n

30    INPUT n
40    sum = sum + n

50    INPUT n
60    sum = sum + n

70    PRINT "Sum is "; sum
```

then you can see that all we are now doing is repeating a pair of BASIC statements:

```
INPUT n
sum = sum + n
```

three times. Incidentally, because we are accumulating a sum by saying 'sum' becomes equal to what it was before plus 'n', we must ensure that we start off with zero in 'sum'. Most computers do not set variables to zero before they are used - the programmer must do this if necessary - hence line 5.

Now we can make the computer obey this pair of statements three times by writing:

```
10   FOR count = 1 TO 3

20        INPUT n
30        sum = sum + n

40   NEXT count
```

or 100 times by writing:

```
10   FOR count = 1 TO 100

20        INPUT n
30        sum = sum + n

40   NEXT count
```

This construction is called a FOR loop and by enclosing a group of statements or a single statement between FOR and NEXT we can make the computer obey a piece of BASIC as many times as we wish. The general form of a simple FOR-NEXT construction is:

FOR | numeric variable | = | start value | TO | stop value

| statements to be repeated |

NEXT | numeric variable |
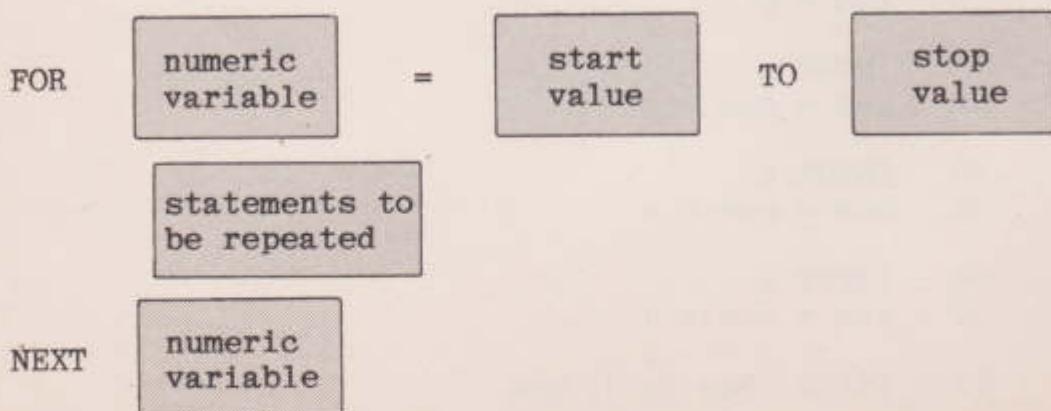
The next program takes as input a list of 30 temperature readings and displays the average of these readings.

```
10    sum =  0

20    FOR  day = 1 TO 30
30       INPUT temp
40        sum = sum + temp
50    NEXT day

60    average = sum/30
70    PRINT "Average temperature is "; average
```

There are 3 points to note

(1) The FOR statement tells the computer to obey the BASIC
    statements between FOR and NEXT a number of times
    specified by the start and stop value.

(2) The numeric variable used after FOR and NEXT is used by
    the computer to keep track of the number of times it has
    executed the loop. Each time it goes round the loop the
    contents of this variable are incremented by one. This
    variable   is usually called the 'control variable'
    because it is used to control the execution of the loop.

(3) The word NEXT is used at the end of the loop to indicate
    its scope, or tell the computer how many statements are
    to be included in the loop.

The next program draws 100 random lines in random colours
(red or yellow).

```
10    MODE 5
20    FOR line = 1 TO 100
30       GCOL 0, RND(2)
40       DRAW RND(1023), RND(1279)
50    NEXT line
60    keypress = GET
70    MODE 7
```

4.2 Use of the control variable

First look at the following program. The input to the
program is the value of a debt, a monthly repayment and an
interest rate. The program is to output a list showing the
outstanding debt after each monthly repayment has been made.

```
10    @% = &2020A
20    INPUT debt, payment, rate
30    FOR month = 1 TO 12
40       debt = debt + debt * rate/100 - payment
50       PRINT "Debt after next payment is "; debt
60    NEXT month
70    @% = &90A
```

The rather strange 'formatting' statement at line 10 is used without detailed explanation. It tells the computer to print numbers with exactly two digits after a decimal point - very useful when handling money. The statement at line 70 switches the PRINT format back to normal. The use of commas in PRINT statements and the use of formatting is described in Appendix 6. You may, however, prefer to ignore the details and simply use the above as a 'recipe' when it is needed.

Given input of 95, 5, 2 this program will produce output:

```
Debt after next payment is 91.90
Debt after next payment is 88.74
Debt after next payment is 85.51
Debt after next payment is 82.22
Debt after next payment is 78.87
Debt after next payment is 75.44
Debt after next payment is 71.95
Debt after next payment is 68.39
Debt after next payment is 64.76
Debt after next payment is 61.06
Debt after next payment is 57.28
Debt after next payment is 53.42
```

As we have already remarked, the control variable is used by the computer to count how many times the statements following the FOR statement have been obeyed. However, there is no reason why the programmer should not also make use of the value of this variable.

The next program is similar to the previous one except that it now uses the value of the control variable inside the loop. The month number is now printed alongside each balance.

```
10    INPUT debt, payment, rate
20    PRINT "        Month          Outstanding debt"
30    FOR month = 1 TO 12
40       debt = debt + debt * rate/100 - payment
50       PRINT month;
60       @% = &02020A
70       PRINT "        ", debt
80       @% = &90A
90    NEXT month
```

Given input of 1256.75, 56.50, 1.25 this program will
generate:

Month	Outstanding debt
1	1215.96
2	1174.66
3	1132.84
4	1090.50
5	1047.63
6	1004.23
7	960.28
8	915.78
9	870.73
10	825.12
11	778.93
12	732.17

We have had to move the two print-formatting statements
inside the loop so that the program keeps switching back to
normal format for printing the month number.

The next program draws a regular polygon. The 'radius',
the number of sides and the centre are supplied as input. If
the number of sides is made very large, the program will
draw a circle. If you are not familiar with trigonometry you
will have to take the details on trust.

```
10      INPUT "Radius", r
20      INPUT "No. of sides", n
30      INPUT "Centre", cx, cy
40      theta = 2*PI/n
50      MODE 5
60      MOVE cx+r, cy
70      FOR side = 1 TO n
80          x = cx + r*COS(side*theta)
90          y = cy + r*SIN(side*theta)
100         DRAW x,y
110     NEXT side
```

Perhaps we should mention that the trigonometric functions
SIN and COS are rather slow, and that there are considerably
more efficient techniques available for drawing circles.

Now although it is common practice to use the value of
the control variable in such contexts as above, you should
never in any circumstances assign to, or attempt to alter
the value of the control variable inside the loop. The
control variable is the record the machine keeps of how many
times it has executed the loop and any interference with
this record can have disastrous consequences.

The practice of changing the control variable to force
termination of a loop tends to obscure the precise nature of
the loop. The same effect can be obtained by using a much

more appropriate construction, a REPEAT loop (see below).

4.3 Non-unit steps in loops

The control variable in a FOR statement can be incremented or decremented by non-unit steps by using the word STEP. For example:

    FOR i = 0 TO 10 STEP 0.2

would cause the body of the loop to be executed 51 times with the control variable taking values:

    0, 0.2, 0.4, 0.6 ... 9.8, 10

The use of this facility is illustrated in the next fragment that prints all the multiples of 3 between 3 and 81.

```
10     FOR i = 3 TO 81 STEP 3
20         PRINT i
30     NEXT i
```

In the next program the increment is input from the keyboard. This program produces a conversion table where each row in the table gives the equivalent in gallons of a quantity in litres. The initial imperial quantity and the final imperial quantity together with the increment in gallons between rows in the table are input from the keyboard.

```
10     INPUT startq, stopq, inc
20     FOR gallons = startq TO stopq STEP inc
30         litres = gallons * 4.546
40         PRINT gallons, litres
50     NEXT gallons
```

The next program plays a chromatic scale, ie it plays all the semitones in an octave.

```
10     FOR pitch = 1 TO 49 STEP 4
20         SOUND 1, -15, pitch, 20
30     NEXT pitch
```

The next program draws an outline of a bar-chart or histogram. The values giving the heights of each bar are given in a DATA statement.

```
10    DATA 200,400,500,700,900,875,850,800,600,150
20    MODE 5
30    y = 100
40    MOVE 0,y : DRAW 1279,y : MOVE 100,y
50    FOR x = 100 TO 1000 STEP 100
60        READ height
70        DRAW x, y + height
80        DRAW x + 100, y + height
90        DRAW x + 100, y
100   NEXT x
```

The general form of a FOR...NEXT construction can thus be extended to:

FOR | numeric variable | = | start value | TO | stop value | STEP | increment value

statements to be repeated

NEXT | numeric variable
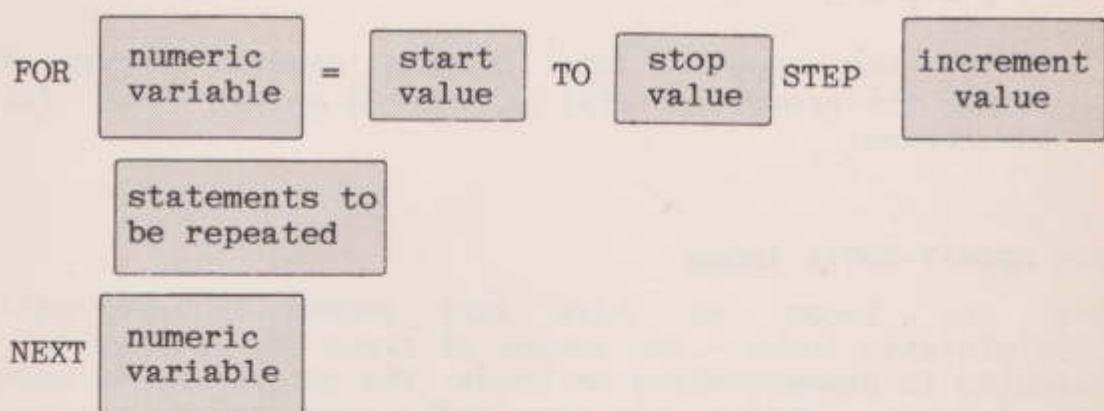
Exercises

1 Write a program that inputs and adds together 15 numbers and then prints their average. Now modify your program so that it inputs an integer n followed by n numbers. The program should display the average of the n numbers.

2 If you have a Model B machine, change the program at the end of Section 4.1 to run in MODE 2 which permits a wider range of colours. (You can use RND(15).)

3 Write a program to print a '4 times table' in the form:

```
1 x 4 = 4
2 x 4 = 8
3 x 4 = 12
   etc.
```

Now modify your program so that it reads an integer, n say, and prints out an "n times table".

4 The statement

```
SOUND 0, -15, f, 50
```

makes different noises on channel 0. The noise made
depends on the value of f which can be anything from 0 to
255. Write a program that makes all the different noises
on channel 0 one after the other. (You will find that
there are in fact considerably fewer than 256 different
noises.)

5 Write a program that plays through the available
frequency range in 'perfect fifths'. The interval of a
perfect fifth contains 7 semitones (or 28 quarter
semitones). Use sound channel 1, 2 or 3.

6 Try the effect of the statement

    r = r + 5

inside the loop of the polygon drawing program. Now
change the program so that it draws a spiral that fills
the screen.

4.4 REPEAT-UNTIL loops

Now the loops we have just looked at are called
deterministic loops - the number of times the loop is to be
executed is predetermined or known. The program knows before
the loop is obeyed how many daily temperatures are to be
averaged or how many monthly balances have to be calculated.
    The next type of loop we are going to look at is a
REPEAT-UNTIL loop. This is used when it is not possible to
calculate, before entering the loop, how many times the loop
is to be executed. We write the program using a REPEAT-UNTIL
structure which means that a statement or group of
statements are REPEATed UNTIL a condition is satisfied. For
example, suppose we wish to use the keyboard as a simple
adding machine: each number we type is to be added to an
accumulating sum. Thus, the statements that are to be
repeated are as before:

    INPUT n
    sum = sum + n

However, this time let's assume that we do not know in
advance how many numbers are to be included in the sum. This
would be the case, for example, with a supermarket checkout
program where the number of items is, of course, variable.
We want the program to keep summing the numbers we type
until we give it a signal to stop - say by typing zero. This
can be summarised by:

```
REPEAT

    INPUT n
    sum = sum + n

UNTIL 0 has been typed
```

This program sums a list of numbers terminated by a zero.

```
 5     sum = 0
10     REPEAT
20       INPUT n
30       sum = sum + n
40     UNTIL n = 0
50     PRINT "Sum is "; sum
```

The general form of such a structure is:

```
  ┌──►REPEAT
  │        ┌─────────────────┐
  │        │ statements to   │
  │        │ be repeated     │
  │        └─────────────────┘
  └────UNTIL    condition
        ┌───────────────────────┐
        │ continue with remainder│
        │     of the program     │
        └───────────────────────┘
```
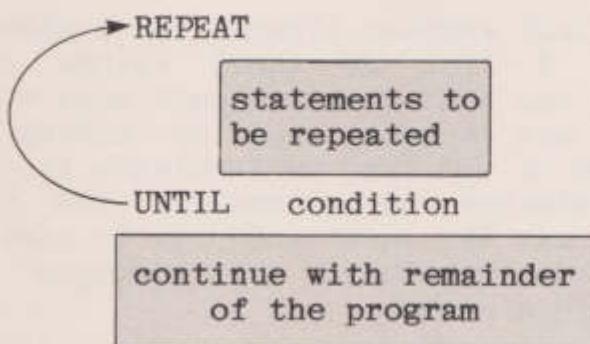
The next program is a trivial example of a computer assisted learning (CAL) program. It repeats an arithmetic problem until the user gives the correct answer. An arithmetic problem such as '16 + 25 = ?' is posed. The user is to input the answer and the program is to check the answer, repeating the question until the answer is correct.

```
10    a = RND(20) : b = RND(20)
20    tries = 0
30    REPEAT
40      PRINT a; "+"; b; "=";
50      INPUT answer
60      tries = tries + 1
70    UNTIL answer = a + b
80    PRINT "Got it! after "; tries; " tries"
```

Here is another variation on this theme. This program keeps setting different random multiplication questions and

reading the answers typed at the keyboard. The program stops when a wrong answer is typed and prints a message indicating how many correct answers were typed.

```
10      questions = 0
20      REPEAT
30         questions = questions + 1
40         a = RND(11) + 1   :   b = RND(11) + 1
50         PRINT a; "x"; b; "=";
60         INPUT answer
70      UNTIL answer <> a*b
80      SOUND 0, -15, 5, 50
90      PRINT "You got "; questions-1; " right"
100     IF questions > 20 THEN  PRINT "Well done!"
```

Remember that the expression 'RND(11) + 1' gives a random number in the range 2 to 12.

4.5 Data terminators

The first program of the last section illustrated a widely used programming structure. A list of input values was processed and the program was such that it could cope with lists of any length. This was accomplished by adding a special code to the bottom of the list to terminate it. We call such a code a data terminator and now examine its use in greater detail. Look at the next program. It adds a sequence of positive numbers together. The sequence is terminated by a single negative number.

```
10      sum = 0
20      INPUT next
30      REPEAT
40         sum = sum + next
50         INPUT next
60      UNTIL next < 0
70      PRINT "Sum is "; sum
```

This program illustrates a common feature of conditional loops: the next value to be processed is obtained at the end of the loop so that this value is tested before it is processed. This necessitates obtaining the first value before entering the loop. In the last section, we cheated by making the data terminator a zero so that it had no effect on the sum.

It is interesting to consider a number of alternative constructions for the loop in the above program, all of which are occasionally produced by beginners and all of which are **wrong**.

```
    sum = 0
    REPEAT
        INPUT next
        sum = sum + next
    UNTIL next < 0
```

This would add the data terminator, which is a negative number, to the total.

```
    sum = 0
    INPUT next
    REPEAT
        INPUT next
        sum = sum + next
    UNTIL next < 0
```

This would not add the first number onto the total and would again add the data terminator onto the total.

```
    sum = 0
    REPEAT
        INPUT next
        sum = sum + next
        INPUT next
    UNTIL next > 0
```

This version adds together the first, third, fifth, etc. numbers in the data and tests the second, fourth, sixth, etc.

If the numbers in the input are being handled in groups of two or more, care has to be taken in handling data terminators. The neatest solution is to insert a group of data terminators which contains the same number of values as the groups into which the rest of the data is organized, as in the next program.

Voting takes place for two political parties in a number of constituencies. The two vote-totals for each constituency are typed in pairs as input to a program and two negative numbers are typed in the input when all pairs of totals have been typed. This program adds up the overall totals for the two parties and reports the overall result.

```
10    party1overall = 0
20    party2overall = 0
30    INPUT party1next, party2next

40    REPEAT
50      party1overall = party1overall + party1next
60      party2overall = party2overall + party2next
70      INPUT party1next, party2next
80    UNTIL party1next < 0

90    PRINT "Party1: "; party1overall
100   PRINT "Party2: "; party2overall
```

If presented with input:

```
3, 7
5, 9
4, 2
-1,-1
```

this program will output

```
Party1: 12
Party2: 18
```

The logical structure of this program is the same as that of the previous one except that the input values are processed in pairs. Two data terminators are needed in order that they can be read by the same statement that reads the other pairs of values. (The first data terminator must be negative and the test for termination of the loop needs to examine only the first value of each pair.) If we wish to type only one data terminator at the end of the input data, a slightly more difficult structure is required for the loop:

```
10    INPUT party1next
20    REPEAT
30      INPUT party2next
40      party1overall = party1overall + party1next
50      party2overall = party2overall + party2next
60      INPUT party1next
70    UNTIL party1next < 0
80    PRINT party1overall, party2overall
```

This version reads only one number at the end of the loop and tests it. Only if this value is not the terminator can a second value be safely read. In addition to being slightly more difficult, this version requires the input values to be typed on separate lines. (Each INPUT statement expects its input values to start on a new line.)

The following program also handles values two at a time. It will play a tune specified in a DATA statement. The values in the DATA statement come in pairs giving the frequency and duration for each note. The sound produced is rather toneless in quality, but we shall see how to improve this in Chapter 10.

```
1    DATA  0,10, 0,10, 28,10, 28,10, 36,10, 36,10, 28,20
2    DATA 20,10, 20,10, 16,10, 16,10, 8,10, 8,10, 0,20
3    DATA  0, 0

10   READ freq, length
20   REPEAT
30       SOUND 1, -15, freq, length
40       READ freq, length
50   UNTIL length=0
```

One problem that will be immediately apparent if you run this program is that consecutive notes of the same pitch will be 'slurred' - there is no gap between notes. A note of zero duration could be inserted between identical notes (see Chapter 1, Section 1.8). Better ways of creating the effect of an inter-note gap are introduced in Chapter 10.

Finally, we present a program in which two REPEAT-UNTIL loops are used one after the other. This program adds up the overall totals for the two parties in an election, but the constituency subtotals for one party are all typed first in the input and are terminated by a negative number. The subtotals for the second party are then typed and are also terminated by a negative number.

```
10    party1overall = 0
20    INPUT party1next
30    REPEAT
40      party1overall = party1overall + party1next
50      INPUT party1next
60    UNTIL party1next < 0

70    party2overall = 0
80    INPUT party2next
90    REPEAT
100     party2overall = party2overall + party2next
110     INPUT party2next
120   UNTIL party2next < 0

130   PRINT "party1: "; party1overall
140   PRINT "party2: "; party2overall
```

The second loop is not encountered until the first loop has

been obeyed the appropriate number of times. The first
REPEAT-UNTIL loop is obeyed until the first negative number
is read and only then can the program carry on to obey the
next loop. The input for the program might consist of the
numbers

```
3  5  4  -1
7  9  2  -1
```

where each number has to be typed on a separate line. The
input could even consist of the numbers:

```
2  3  7  9  4  -1
1  7  6  -1
```

if candidates for party 1 are standing for election in more
constituencies than are candidates for party 2.

4.6 Use of logical variables in loops

A logical variable often provides a neat way for a
programmer to express the terminating condition for a loop,
and to test subsequently why the loop was terminated.
Consider the following program which reads input consisting
of 365 pairs of values, one pair for each day of the year.
The first number in each pair is the average temperature for
the day and the second is the number of hours of sunshine
for the day. The program counts how many days elapsed before
the first day on which both the average temperature exceeded
15 degrees and the number of hours of sunshine exceeded 10.
The program caters for the possibility that there is no such
day.

```
10    day = 0
20    warmdayfound = FALSE

30    REPEAT
40      day = day + 1
50      INPUT nextemp, nextsun
60      warmdayfound = nextemp > 15 AND nextsun > 10
70    UNTIL warmdayfound OR (day = 365)

80    IF warmdayfound THEN
         PRINT "There were ";
                day - 1; " days before the 1st. good day."
      ELSE   PRINT "A bad year!"
```

More realistically, the data for this program would be held
on a file and input from there (see Appendix 2).

Exercises

1 An organisation has approximately $10000 available to be allocated in small amounts to approved charities. The amounts approved for each charity are to be typed as input for a program in the order in which the requests are received. Write a program that will read in these amounts and report as soon as over $10000 has been allocated.

2 In the first CAL program of Section 4.4, the output produced by the programs is occasionally a little ungrammatical. Correct this.

3 Write a program that accepts a person's initial bank balance followed by a sequence of positive and negative real numbers representing transactions. A positive number represents a credit entry in the account and a negative number represents a debit entry. The input is terminated by a zero entry. The program should print the new balance.

4 Write a program that switches to MODE 5 and then draws a picture under the control of a user seated at the keyboard. The user will repeatedly type a pair of x-y coordinates telling the program to which point it should next draw a line. The program should terminate and switch back to MODE 7 when two negative values are typed. The function TAB can be used to keep the cursor out of the way of the picture being drawn. For example,

```
    PRINT TAB(0,0); "                    " ; TAB(0,0);
```

moves the character cursor to the top left-hand corner of the screen and wipes out any numbers already displayed at the start of the first line of the screen. This could be done by the program before input of each pair of numbers.

4.7 Timing delays

A timing element is used in many contexts in computer programming. Measuring time and acting upon particular timing signals is a feature of real-time programming. In using your computer there are many contexts in which you might make timing measurements. For example: as an element in games; as a delay in animation (see Chapter 11); or in simply slowing the rate at which a continuous sequence or list of information is displayed on the screen. Look again at the first program in Section 4.2. Suppose we wanted a 1 second delay between the appearance of each new line on the screen. We could proceed as follows:

```
20      FOR month = 1 TO 12
30          debt = debt + debt*rate/100 - payment
40          PRINT "debt after next payment is ", debt
50          ****  delay proceedings for 1 sec. ****
60      NEXT month
```

Now we can set up a delay in two ways. Firstly we can use an empty FOR statement:

```
50      FOR i = 1 TO 1400 : NEXT i
```

We know that the BBC computer takes approximately 1 second to execute this empty loop. Although the loop is not doing anything the computer has no knowledge of this and still has to set up its loop housekeeping operations. (Incidentally we have now introduced a loop within a loop. This structure is examined in more detail in Chapter 5.)

Secondly we can use the function TIME. You can imagine TIME as a numeric variable that can be initialised and is then incremented in units of 1/100th of a second. Thus a 1 second delay using this facility would be obtained by

```
50      TIME = 0 :   REPEAT : UNTIL TIME > 100
```

Timing delays can also be obtained by using the INKEY function which is described in later chapters.

Exercises

1 Extend the polygon drawing program of Section 4.2 so that it waits for 30 seconds after completing its pattern and then switches back to MODE 7.

2 Extend the second CAL program of Section 4.4 so that it stops either when a wrong answer is typed or when a maximum time limit has been exceeded.

Chapter 5 **Statements within statements**

Sometimes a control structure needs to be included inside another. For example it is possible to have a conditional statement inside a loop, a loop within a conditional statement and loops within loops. In this chapter we demonstrate how to write programs involving some of the most commonly occurring nested control structures.

5.1 IF statements within loops

We have seen in Chapter 3 that a FOR statement can take the form

FOR ▢ numeric variable ▢ = ▢ start value ▢ TO ▢ stop value ▢

▢ statements to be obeyed ▢

NEXT ▢ numeric variable ▢
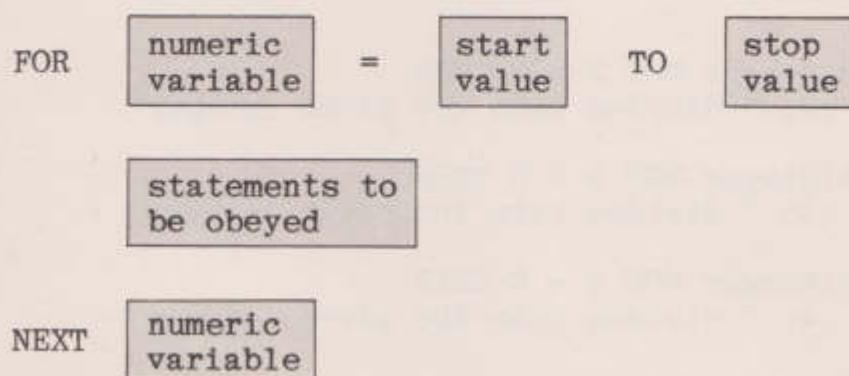
A statement to be obeyed repeatedly can in fact be any BASIC statement. We now illustrate the case where an IF statement is obeyed repeatedly by writing a program to print all the integers between 2 and 9 which divide exactly into a given integer. We can describe in outline what the program must do as follows:

```
10   INPUT giveninteger
20   FOR i = 2 TO 9
```

▢ statement to test whether i divides exactly into the given integer ▢

```
40   NEXT i
```

In order to test whether an integer 'i' divides exactly into 'giveninteger' we can use

```
30    IF giveninteger MOD i = 0 THEN
          PRINT ;i; " divides into the given integer"
```

and this is the statement that must follow the above FOR statement. So putting them both together we get the following program.

```
10    INPUT giveninteger
20    FOR i = 2 TO 9

30        IF giveninteger MOD i = 0 THEN
              PRINT ;i; " divides into the given integer"

40    NEXT i
```

The FOR statement in the above program will behave as if a sequence of eight separate IF statements were obeyed:

```
IF giveninteger MOD 2 = 0 THEN
  PRINT ;2; " divides into the given integer"

IF giveninteger MOD 3 = 0 THEN
  PRINT ;3; " divides into the given integer"

IF giveninteger MOD 4 = 0 THEN
  PRINT ;4; " divides into the given integer"

        .
        .
        .
IF giveninteger MOD 9 = 0 THEN
  PRINT ;9; " divides into the given integer"
```

Thus, given input of 18, the program will print

```
2 divides into the given integer
3 divides into the given integer
6 divides into the given integer
9 divides into the given integer
```

The next program demonstrates one possible way of making the computer play a major scale. The following table shows the intervals between the notes of the scale of C major.

	Sol-fa notation	Note name	Interval in semitones	Interval in quarter semitones
1	Doh	C		
			2	8
2	Ray	D		
			2	8
3	Me	E		
			1	4
4	Fah	F		
			2	8
5	Soh	G		
			2	8
6	Lah	A		
			2	8
7	Te	B		
			1	4
8	Doh	C		

Middle C corresponds to frequency code number 53. Thus the statement

    SOUND 1, -15, 53, 20

plays Middle C for about one second. To play the scale of C major, we could use

```
10    pitch = 53
20    FOR note = 1 TO 8
30        SOUND 1, -15, pitch, 20
40        IF note=3 OR note=7 THEN  pitch = pitch + 4
                             ELSE  pitch = pitch + 8
50    NEXT note
```

The next program includes an IF statement within a REPEAT-UNTIL loop. It is an elaboration of one of the computer assisted learning programs of the last chapter. The program is now made more realistic by including within the REPEAT loop an IF statement that allows an appropriate message to be printed.

This program also illustrates another common problem in loop programming. Frequently, when a loop is terminated, the reason for the termination must be known so that it can be used, for example to print an appropriate message. Using logical variables - or state variables as they are technically known - gives a neat way of doing this.

```
10    tries = 0
20    a = RND(100) : b = RND(100)
30    REPEAT
40        PRINT a; " + "; b; " = ";
50        INPUT answer
60        correct = (answer = a + b)
70        IF NOT correct THEN PRINT "No!"
80        tries = tries + 1
90    UNTIL correct OR tries = 3

100  IF correct THEN
         PRINT "Got it after "; tries; " tries."
     ELSE
         PRINT "Sorry you have had 3 tries." :
         PRINT "The correct answer is "; a + b
```

The next program illustrates the use of several IF statements inside a REPEAT loop. It also uses the function GET$ which 'gets' the next character typed at the keyboard without the character being displayed on the screen.

This program acts as a simple graphics 'sketchpad' under the control of a user seated at the keyboard. (Compare this program with the exercise set at the end of Section 4.6.) The user inputs a start position (x-y coordinates) and then draws a picture by repeatedly typing a character "N", "S", "E" or "W". Each character typed by the user causes the line drawn by the program to be extended a short distance in the direction indicated (North, South, East or West). The program terminates when the character "Q" for Quit is typed.

```
10    INPUT "Start position", x, y
20    MODE 5
30    MOVE x, y

40    REPEAT
50        c$ = GET$
60        IF c$ = "N" THEN y = y + 5
70        IF c$ = "S" THEN y = y - 5
80        IF c$ = "E" THEN x = x + 5
90        IF c$ = "W" THEN x = x - 5
100       DRAW x,y
110   UNTIL c$ = "Q"

120   MODE 7
```

Exercises

1 Given input consisting of 30 numbers, some of which are positive and some negative, write a program that finds

the average of the positive numbers and the average of the negative numbers.

2   The Scottish bagpipe scale starts on the A above Middle C (pitch code number 89). The intervals between successive notes (in quarter semitones) are approximately 8, 6, 6, 8, 6, 6, 8. Incidentally, these notes are not all available on a piano. Write a program that plays a Scottish bagpipe scale.

3   Another way of playing a scale would be to insert the 8 required pitch values in a DATA statement. The scale could then be played with a simple FOR loop. Do this.

4   Modify the CAL program of the last section so that it times the user and prints an appropriate message telling him how long it took him to find the correct answer.

5   Extend the sketchpad program so that the line can be switched on or off by means of two other keys. One way of doing this would be to use a logical variable in which the program keeps a record of whether the line is on or off. Another way would be to switch colours using GCOL. Switching the line off would correspond to making the current plotting colour the same as the background colour (GCOL 0,0).

When the line is off, you will find that you have no idea where on the screen you are. You will be able to deal with this problem when you have read Chapter 9.

5.2 Loops within loops

One loop inside another loop is a nested structure frequently encountered in programs. This is because so much data analysed by computer programs is organised in the form of tables.

Consider the problem of processing the 5 examination marks obtained by each of 25 candidates. The structure required is a loop of the form

FOR candidate = 1 TO 25

> process the marks obtained
> by the candidate

NEXT candidate

Processing one candidate's marks might involve reading the 5 marks obtained by the candidate, computing his average and testing for a pass or fail:

```
10      total = 0
20      FOR exam = 1 TO 5
30        INPUT mark
40        total = total + mark
50      NEXT exam

60      average = total/5
70      IF average >= 50 THEN
          PRINT "Passed! Average is "; average
        ELSE
          PRINT "Failed! Average is "; average
```

and a segment of program like this must be obeyed 25 times.
This is achieved simply by inserting the the above fragment
in its entirety into the 'candidate', loop making a loop
within a loop structure:

```
5      FOR candidate = 1 TO 25
```

```
10      total = 0
20      FOR exam = 1 TO 5
30        INPUT mark
40        total = total + mark
50      NEXT exam

60      average = total/5
70      IF average >= 50 THEN
          PRINT "Passed! Average is "; average
        ELSE
          PRINT "Failed! Average is "; average
```

```
80     NEXT candidate
```

Incidentally, in practice, input would of course be taken
from a file (see Appendix 2). In this particular example can
you see what the difference would be, as far as the screen
display is concerned, between file input and keyboard input?

In the above program the outer loop is executed 25 times.
Once the computer enters the innermost loop, it stays there
until this loop has been executed 5 times. The innermost
pair of instructions at lines 30 and 40 are executed a total
of 25x5 times and 125 marks are supplied as input in the
form of 25 groups of 5 marks. If we imagine a counter
associated with each loop, then the 'exam' counter would be
turning 5 times as fast as the 'candidate' counter,
resetting to 1 for each new candidate.

```
outer loop
candidate      1

        inner loop
              exam      1
                        2
                        3
                        4
                        5

candidate      2

        inner loop
              exam      1
                        2
                        3
                        4
                        5

              .
              .
              .
candidate     25

        inner loop
              exam      1
                        2
                        3
                        4
                        5
```
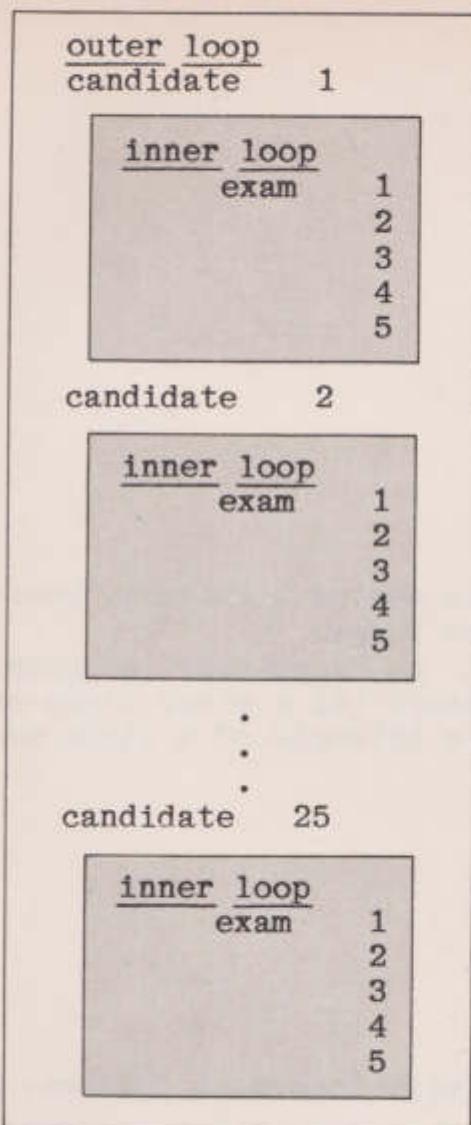
To improve your understanding of the idea of a nested loop, you should examine the difference in behaviour between the following two fragments of program.

```
10      FOR i = 1 TO 3
20        FOR j = 1 TO 3
30          PRINT i,j
40        NEXT j
50      NEXT i
```

prints

```
            1           1
            1           2
            1           3
            2           1
            2           2
            2           3
            3           1
            3           2
            3           3
```

```
10    FOR i = 1 TO 3
20      PRINT i
30    NEXT i
40    FOR j = 1 TO 3
50      PRINT j
60    NEXT j
```

prints

```
1
2
3
1
2
3
```

The first fragment is of course a nested structure, whereas the second is just two consecutive loops.

As a further illustration, let us construct a program that will display a triangle of stars (of a given size) on a background of dots. For example a triangle of 4 lines would be:

```
..........
....*.....
...***...
..*****..
.*******.
..........
```

In outline, the program required will behave as follows:

```
10    INPUT noofrows
20    *** calculate width of picture ***

30    FOR ch = 1 TO widthofpic : PRINT "."; : NEXT ch
40    PRINT

50    FOR rowno = 1 TO noofrows

         *** print the next row of the triangle with    ***
         *** the appropriate no. of dots on either side ***

60    NEXT rowno

70    FOR ch = 1 TO widthofpic : PRINT "."; : NEXT ch
```

The width of the picture is given by '2*noofrows + 1'. Printing a row of the triangle can be described in more detail as:

```
      Calculate number of stars in this row
      Calculate number of background dots at each end of  row
      FOR ch = 1 TO noofdots : PRINT "."; : NEXT ch
      FOR ch = 1 TO noofstars: PRINT "*"; : NEXT ch
      FOR ch = 1 TO noofdots : PRINT "."; : NEXT ch
```

The number of stars to be printed on a row is '2*rowno - 1'
and   the   number   of   background   dots   required   is
'noofrows + 1 - rowno'. These loops must be nested inside
the second loop in the outline above because our triangle is
made up of a number of such lines. Filling in these details
we get the following program:

```
10    INPUT noofrows
20    widthofpic = 2*noofrows + 1

30    FOR ch = 1 TO widthofpic : PRINT "."; : NEXT ch
40    PRINT

50    FOR rowno = 1 TO noofrows
51        noofstars = 2*rowno - 1
52        noofdots  = noofrows + 1 - rowno
53        FOR ch = 1 TO noofdots : PRINT "."; : NEXT ch
54        FOR ch = 1 TO noofstars: PRINT "*"; : NEXT ch
55        FOR ch = 1 TO noofdots : PRINT "."; : NEXT ch
56        PRINT          :REM to start a new line.
60    NEXT rowno

70    FOR ch = 1 TO widthofpic : PRINT "."; : NEXT ch
```

The final example in this section involves an   IF
statement inside a REPEAT-UNTIL loop inside a FOR loop.
Part of a daily sales analysis program involves printing
a list of the number of items costing more than $10 that
were sold in each of 63 departments:

```
FOR dept = 1 TO 63
```

```
    analyse the sales for one department
```

```
NEXT dept
```

The input for each department consists of a list containing
the price of each item sold in the department on the day in
question,   and   each   list   is terminated by -1. One
department's sales can therefore be analysed by:

```
20    largesales = 0
30    INPUT nextitemprice

40    REPEAT
```

> test 'nextitemprice' to see
> if it is a large sale

```
60        INPUT nextitemprice
70    UNTIL nextitemprice < 0
```

Filling in the rest of the details we obtain:

```
10    FOR dept = 1 TO 63
20      largesales = 0
30      INPUT nextitemprice

40      REPEAT
50        IF nextitemprice >= 10 THEN
              largesales = largesales + 1
60        INPUT nextitemprice
70      UNTIL nextitemprice < 0

80      PRINT "Dept no. "; dept; " has made ";
              largesales; " largesales"
90    NEXT dept
```

Exercises

1 Write a program that accepts, as input, two numbers
representing the width and height of a rectangle. The
program is then to print such a rectangle made up of
asterisks, for example:

```
*********
*********
*********
*********
```

2 Write a program to accept a single integer and print a
multiplication table for all the positive integers up to
the one specified as input.

5.3 Nested IF statements

Consider the following sequence of three IF statements:

```
10    IF age < 21   THEN PRINT "Refuse policy"
20    IF age >=35   THEN PRINT "Issue policy with discount"
30    IF age >=21 AND age < 35 THEN
                        PRINT "Issue policy at full price"
```

Here three consecutive IF statements are used to select one of three possible courses of action. The three conditions used are such that one and only one of them must be TRUE. However, even if the condition in the first IF statement is TRUE, the computer will still waste time testing the conditions in the remaining two IF statements. If the conditions in the first two IF statements are FALSE, the condition in the third must be TRUE and the computer will again waste time testing it. These inefficiencies can be eliminated by using a more appropriate IF statement structure for making the tests involved. Let us start by noting that IF age < 21 then the first write statement should be obeyed and no further tests made. This can be achieved by using an IF-THEN-ELSE structure which can be outlined as follows:

```
10    IF age < 21 THEN PRINT "refuse policy"
      ELSE
```

```
            statement to be obeyed
            only if age >= 21
```

Any statement inserted after the ELSE will be obeyed only if age >= 21. In this example we can obtain the effect we require by making the statement after the ELSE a further IF statement which distinguishes the cases age >= 35 and age < 35:

```
10    IF age < 21 THEN PRINT "Refuse policy"
      ELSE
            IF age >= 35 THEN PRINT "Issue with discount"
                        ELSE PRINT "Issue at full price"
```

Note that this is a single statement as far as BBC BASIC is concerned and has only one line number. This version of the program will not test the second condition if the first is satisfied and will automatically obey the third PRINT statement if the first two conditions are FALSE. We can

illustrate the behaviour of this nested IF statement as
follows:

```
                        age < 21?
          TRUE             │             FALSE
            │                              │
            ▼                              ▼
     refuse policy              age >= 35?
                        TRUE       │       FALSE
                          │                  │
                          ▼                  ▼
                   issue  policy      issue  policy
                   with discount      at full price
```
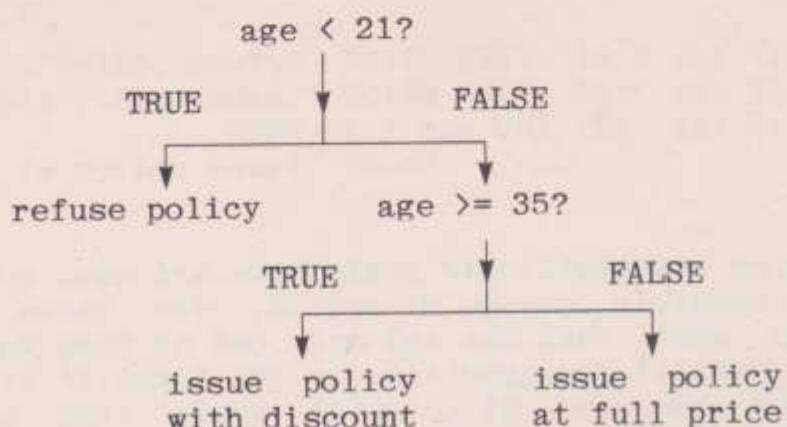
We can imagine the computer following one path from the top
of this diagram and carrying out the action at the end of
that path.

Unfortunately, the extent to which we can nest IF
statements in BBC BASIC is limited in several ways. For
example, unexpected effects can be obtained if we use an IF-
THEN-ELSE after the THEN of another IF statement. (You will
find that the computer can not decide to which IF the ELSE
belongs.) We are also limited by the restriction that our
complete nested IF statement must fit into 240 characters (6
lines in MODE 7). We suggest that the use of nested IF
statements is limited to the use of IF after ELSE as
illustrated above.

The standard way of implementing more complex IF
structures in other BASIC dialects is to use the GOTO
statement. However, the techniques introduced in Chapter 7
will enable us to program complex nested control structures
without resorting to the use of GOTO.

Exercises

1 Write a program that inputs two person's names, each name
being followed by their age. The program should output a
message stating whether they are the same age and, if
not, it should state who is older.

5.4 Stepwise refinement

In writing some of the programs in this chapter, we have
informally introduced the technique that is sometimes termed
'stepwise refinement' or 'top-down program design'. Instead
of attempting to write down a complete BASIC program in one
step, we first decide what the outermost control structure
in the program should be. For example, in writing our sales
analysis program, we could see immediately that there were

63 separate sales analyses to be made. Giving a brief
English description to the process of analysing one
department's sales enabled us to write down an outline of
this loop:

```
FOR dept = 1 TO 63
```

> analyse the sales for one department

```
NEXT dept
```

Having got this clear in our minds, we then concentrated on
the rather easier sub-problem of analysing one department's
sales and the program for doing this was eventually inserted
in the above structure. This sub-problem was of course
tackled by a further application of stepwise refinement.

Such an approach enables the construction of a complex
nested control structure to be broken down into a number of
simpler programming tasks that are more or less independent
of each other.

As another example, we may require a computer assisted
learning program that repeatedly sets a new problem until
the user of the program indicates that he wishes to stop.
The program should always wait until a correct answer to the
last problem has been input before going on to the next one.
The section of program for setting a problem and processing
the users attempts at an answer may be quite complicated,
but before considering the details of this process we must
plan the overall structure of the program as follows:

```
REPEAT
```

> Set a problem and process the answer
>   (repeating the problem if necessary)

```
INPUT "More? Y/N:" reply$
```

```
UNTIL reply$ = "N"
```

We are now free to concentrate on the details of setting a
single problem and processing the answer. A version of this
sub-problem has already been dealt with in Section 5.1 and
the present program can be completed by inserting the
previous solution at the appropriate place in the above
outline:

```
 5     REPEAT
10        tries = 0
20        a = RND(100) : b = RND(100)
30        REPEAT
40           PRINT a; " + "; b; " = ";
50           INPUT answer
60           correct = (answer = a + b)
70           IF NOT correct THEN PRINT "Wrong!"
80           tries = tries + 1
90        UNTIL correct OR tries=3

100       IF correct THEN
              PRINT "Got it after "; tries; " tries."
          ELSE
              PRINT "Sorry you have had 3 tries." :
              PRINT "The correct answer is "; a + b

110       INPUT "More? Y/N:" reply$
120    UNTIL reply$ = "N"
```

Stepwise refinement is just one aspect of the set of programming techniques known as 'structured programming'. Stepwise refinement is discussed further in Chapter 7 when we introduce techniques for giving names to different sections of a program.

Chapter 6 **Handling lists**

If a program systematically processes a collection of variables, it may not be convenient for the programmer to give each of these variables a different name. For example, it would be rather tedious to write:

```
totalprice = priceofhammer + priceofsaw    +
             priceofaxe    + priceofplane  +
             priceofchisel + priceofvice   +
             priceofscrewdr+ priceofspanner
```

Instead of giving such a group of variables separate names, it is often more convenient to give them a collective name and to refer to the individual variables in the collection by a reference number or 'subscript'.

6.1 One-dimensional arrays

A one-dimensional array is a set of storage locations, or variables, all of the same type, which share the same name but have different subscripts. For example the information:

Name of variable	Contents
priceofhammer	5.77
priceofsaw	3.15
priceofaxe	2.50
priceofplane	16.33
priceofchisel	2.50
priceofvice	13.45
priceofscrewdr	.86
priceofspanner	1.98

could be stored in a one-dimensional array:

Name of variable	Contents
priceof(1)	5.77
priceof(2)	3.15
priceof(3)	2.50
priceof(4)	16.33
priceof(5)	2.50
priceof(6)	13.45
priceof(7)	.86
priceof(8)	1.98

The very first thing we have to do with this type of storage facility is to tell the computer that we are going to use it and how long our list or lists will be. The computer needs to set aside enough space for all the variables in a list.

Suppose we had 1000 item prices to store. Right at the beginning of the program we write

    DIM   priceof(1000)

DIM is short for dimension and this statement informs the computer that we want 1000 variables all referred to by the same name - 'priceof'. In computer parlance such a list is called a one-dimensional array and the individual boxes are called array elements. Now, from within a program, array elements can be treated as ordinary variables. For example we could write:

    PRINT "The price of item 2 is "; priceof(2)

and the computer would print out the contents of the second variable in the list. Similarly we could write:

    PRINT "The price of item 57 is "; priceof(57)

and the computer would print out the value of the 57th variable in the list.

Here is an example that performs a simple calculation using array elements.

    PRINT "Price difference between items 5 and 12 is ";
    PRINT ;ABS (priceof(5) - priceof(12))

This simply prints out the difference between the values of the 5th and 12th variables in the list. Actually, the DIM statement reserves storage locations numbered from zero upwards, but in many situations it is convenient, and less confusing, to ignore location zero.

These examples show how we set up a list and how

individual elements can be treated as single variables. We now move on to the more general considerations involved in using such lists.

6.2 Systematic and random access to array elements

One point of having an array facility is that it is easy to carry out the same operation on all the elements in the list. For example, if we wanted to find the total cost of all the items in 'priceof' we would write:

```
110    cost = 0
120    FOR item = 1 TO 1000
130        cost = cost + priceof(item)
140    NEXT item
150    PRINT "Cost of all items is "; cost
```

Here we have used a deterministic loop to systematically process all the elements in the array. The control variable 'item' varies from 1 to 1000 and 'priceof(item)' refers successively to

```
priceof(1)
priceof(2)
priceof(3)
      .
      .
      .
priceof(1000)
```

as the loop progresses. Line 30 is thus a general statement specifying that the contents of an array element are to be accumulated into 'cost'. The particular element accessed is determined by the value of the control variable, which of course is incremented by one each time through the loop. We thus have a facility for referring to consecutive elements in the array. It is important to note that each array element has two quantities associated with it:

(1) its reference number or subscript, and
(2) its contents.

The next program is a simulation of a catalogue system which, in its simplest form, might be a list of prices and implied reference numbers: 1,2,3,...etc. It inputs 1000 prices from a file (see Appendix 2) to be stored in an array. The program then continually loops requesting a reference number in the range 1 to 1000. On receipt of the reference number it is to type out the corresponding price. The loop terminates when a stop code is typed.

94

```
10    DIM priceof(1000)
20    prices = OPENIN ("pricelist")
30    FOR item = 1 TO 1000
40      INPUT# prices, priceof(item)
50    NEXT item
60    CLOSE# prices

70    INPUT ref
80    REPEAT
90      PRINT "Price of item "; ref; " is "; priceof(ref)
100     INPUT ref
110   UNTIL ref = 0
120.  PRINT "You have terminated the program."
```

Now in this program there are two loops. The first, lines 30
to 50, is a deterministic loop systematically going through
the array. It is equivalent to writing

```
INPUT# priceof(1)
INPUT# priceof(2)
INPUT# priceof(3)
        .
        .
        .
INPUT# priceof(1000)
```

and this loads 1000 prices from the input file into
consecutive array elements.
    In practice this is a program that may run all day in a
warehouse until a stop code (zero) is typed. The first loop
is obeyed as soon as the program is started and takes input
from a magnetic disk or cassette file. The program would
normally be waiting inside the second loop at line 90 for a
'ref' number to be typed.
    When a reference number is typed the computer prints out
the contents of the corresponding array element. This is
called random access because the computer accesses any
element out of the 1000 depending on which number is input
to the numeric variable 'ref'. This is illustrated in the
diagram below. Any number in the range 1 to 1000 is typed
into the numeric variable 'ref'. The computer then prints
out the contents of 'priceof(ref)'. The contents of 'ref'
point to the desired array element. This is an extremely
important facility in computer programming and is used
extensively in data structures and databases.

priceof
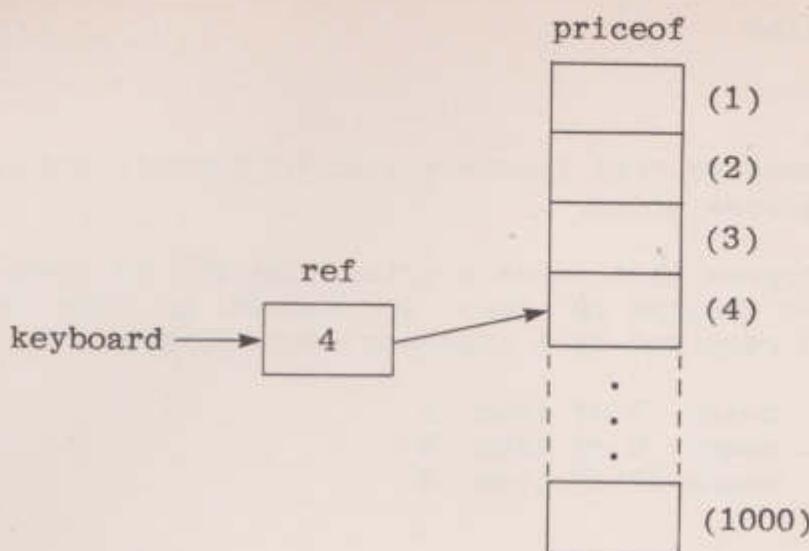
6.3 Subscript range

In the above example if you input a number from the keyboard that is less than 0 or greater than 1000 then the program will fail and execution will terminate with an error message. This is because the computer detects that you have tried to access an element that does not exist. You will find that when you start working with arrays this is one of is the commonest errors you will make.

It is always good programming practice to check that subscripts typed from a keyboard are in range. It is true that if you attempt to use an out of range subscript, then this will be trapped by the computer and an execution error will occur. However there are two consequences of this:

(1) The program will terminate unexpectedly.

(2) If you are writing the program for use by a third party, who may not necessarily know a lot about computers, then a subscript error message followed by program termination could be baffling.

To make the program more robust we could write:

```
70     INPUT ref
80     REPEAT
90         IF ref<0 OR ref>1000 THEN  PRINT "Invalid code"
           ELSE  PRINT "Price of item "; ref;
                          " is "; priceof(ref)
100        INPUT ref
110    UNTIL ref = 0
```

The program is not yet completely foolproof: what happens if zero is typed as soon as the program is obeyed? How should

this dealt with?

Exercises

1 Write a program that inputs a list of numbers and prints them in reverse order.

2 Write a program that reads a price list for 10 items from a file. The program is then to accept a list of 10 quantities required by a customer, for example:

```
5         means  5 of item  1
0         means  0 of item  2
12        means 12 of item  3
          .
          .
          .
27        means 27 of item 10
```

The program should calculate the total cost of the customer's order.

3 As Exercise 2 but this time the program is to accept a list of pairs where each pair contains an item number followed by the quantity required. The item numbers appear in any order and the list can contain any number of orders, for example:

```
10, 2     means 10 of item 2
25, 4     means 25 of item 4
 3, 2     means  3 of item 2
          .
          .
          .
65, 9     means 65 of item 9
 0, 0     means the end of the list.
```

The program should check that each item number typed does in fact exist.

4 Write a program to quote, in weeks and days, the duration between two specified dates. Both dates are in the same year and each date is in the form of two integers: the first is the day and the second is the month. The dates are supplied as input. The numbers of days in each month of the year should be included in a DATA statement. (These DATA values should be read by the program into an array.) Assume to start with that February always has 28 days. Your program should check that both dates are valid and are supplied in the correct order.

Once the program is working, you could extend it to input the number of the year, work out if it is a leap year, and possibly update its record of the number of days in February.

6.4 String arrays and simple databases

Elsewhere we have introduced string variables and the use of such variables in simple string processing problems. More complex operations involving strings often require the use of arrays of strings. We have already met numeric arrays and string arrays embrace the same concept. An array of strings is a list of entities all having the same name and referred to individually using a subscript notation. However, since strings can be of any length, up to 255 characters, the elements of a string array can be of unequal length. For example:

```
                            name$

   name$(1)        | screwdriver   |
                   |_____    |
   name$(2)        | saw       |
                   |_____      |
   name$(3)        | axe   |
                   |___ . __|
                     . 
                     . 
                     . 

   name$(100)      | brick hammer  |
                   |_____|
```
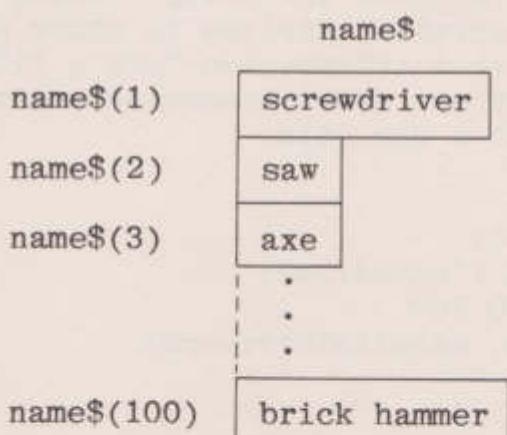
Realistic programs containing arrays of strings usually require vast amounts of data to initialise them. We may have, for example, a name and address list comprising thousands of entries and this must be the commonest use of string arrays in computers, judging by the amount of mail (junk and otherwise) that is addressed using computer printed addresses. Typical processing operations might be to extract from the list all those addresses containing the same postal code, or all those sharing the same town, etc. Systems like this that can be processed or interrogated are usually called databases. A database could be a simple list, or a series of lists, or the information may be structured in a particular way using combinations of arrays. How complex database structures are set up is outside the scope of this text, but we can look at very simple databases.

A simple database can be initialised by reading the contents of a file, usually from disk. Thus we may have, for example, in a department store, a stock database held on a large file. The data is read into the program data structure at 9 a.m., say, and then used all day long by operators. During the course of a day new information may be added to the database or existing information altered. Thus we may have a general program structure:

```
        initialize data structures from file

        REPEAT
            wait for information from user
            process information  from user
        UNTIL endofday

        write contents of data structure back
        to file
```

We have already seen a simple example of this program structure in the last section. We will now look at how we can set up simple data structures involving string arrays. The next program uses an array of strings to store a list of names and addresses. We read information from a file into a single string array, carry out some processing and write the processed information into a new file.

```
10      DIM namelist$(100)
20      innames = OPENIN ("namefile")
30      FOR record = 1 TO 100
40        INPUT# innames, namelist$(record)
50      NEXT record
60      CLOSE# innames

70      outnames= OPENOUT ("extracted")
80      INPUT city$
90      FOR record = 1 TO 100
100       IF INSTR(namelist$(record), city$) <> 0
          THEN PRINT# outnames, namelist$(record)
110     NEXT record
120     CLOSE# outnames
```

(The string function INSTR is discussed fully in Chapter 8. It is used here to test whether the string that is its first parameter contains the string that is its second parameter.) In this simple program the writing and processing are carried out simultaneously. The next data structure is a little more complicated, consisting of four parallel arrays making up a stock control database. For example:

refno	name$	stock	priceof
157	screwdriver	352	3.75
163	hammer	27	5.62
175	spanner	31	1.35
181	axe	4	7.23
:			
:			

This is a simple example of a database. We may want to interrogate the database so that it prints out all the screwdrivers in stock:

```
refno. 157 (screwdriver) 352 in stock
refno. 239 (screwdriver)  28 in stock
refno. 575 (screwdriver)  37 in stock
```

Alternatively we may request the price of a certain item by giving the database a reference number:

```
The price of refno. 157 (screwdriver) is 3.75
```

We may want to give the database the information that a quantity of particular items have been sold. All these operations can be carried out easily on a simple data structure comprising four 'parallel' arrays as above. The array 'refno' could be either a numeric array or a string array depending on the operations that are to be carried out. The array 'name$' must be a string array and 'stock' and 'priceof' must be numeric arrays. Let's look at the first operation - printing out all the items belonging to a particular class, their associated reference numbers and the quantity in stock.

```
10      DIM refno(100),name$(100),stock(100),priceof(100)
            .
            .
        Statements to initialize arrays from file
            .
            .
510     INPUT "Name of item", class$
520     FOR i = 1 TO sizeofstock
530       IF name$(i) = class$ THEN
            PRINT "refno "; refno(i); " (";
                  name$(i);") "; stock(i); " in stock"
540     NEXT i
```

In order to give the price associated with a reference number we could use

```
610     INPUT "Stock number", number
620     i = 0
630     REPEAT
640       i = i + 1
650     UNTIL number = refno(i)
660     PRINT "Price of ref. no "; refno(i); " (";
                  name$(i); ") is "; priceof(i)
```

To process an order and update the stock quantity:

```
710      INPUT "Stockno", number, "Quantity", quantity
720      i = 0
730      REPEAT
740         i = i + 1
750      UNTIL number = refno(i)
760      IF stock(i)-quantity<0 THEN
            PRINT "not enough in stock"
         ELSE  PRINT "Cost: "; quantity*priceof(i) :
            stock(i) = stock(i) - quantity
```

In the next chapter, we shall see how a section of program can be defined as a named 'procedure'. In a complete stock processing program, we would define each of the above sections of program as a named procedure and the main part of the program would organise the 'calling' of these procedures in response to requests from a user of the program. When the program recognises an 'out of stock' condition we could call another procedure that writes to a re-order file. In the next chapter, we shall revisit this example and see how to select such procedures using a standard 'menu selection' approach.

The next example uses four string arrays and also involves calculating element addresses. The program accesses a simple personal database. This is in the form of a string array containing friends' names and a set of corresponding arrays containing such attributes as, for example, date of birth, telephone number, political affiliations. The program accepts input in the form

Mary, birthday

and this causes an access via the name Mary to the corresponding element in the array storing birthdays. The appropriate output would be:

Mary's birthday is 22:7:65

The arrays and their contents might be as follows:

	name$	telephone$	birthday$	politics$
	John	051-44567	13:7:60	Trot
	Clare	665787	1:10:58	SDP
Mary →	Mary	345678	→ 22:7:65	SDP
	Jill	98-56455	3:1:46	SDP
	Nigel	657865	4:8:64	Gaylib

The program is as follows.

```
10    DIM name$(100), telephone$(100), birthday$(100),
          politics$(100)
                .
                .
      initialize arrays from a file
                .
                .
40    INPUT firstname$, attribute$
50    i = 0
60    REPEAT
70      i = i + 1
80    UNTIL name$(i) = firstname$
90    position = i

100   IF LEFT$(attribute$,1) = "t" THEN
          PRINT firstname$; "'s "; attribute$;
                  " is "; telephone$(position)
110   IF LEFT$(attribute$,1) = "b" THEN
          PRINT firstname$; "'s "; attribute$;
                  " is "; birthday$(position)
120   IF LEFT$(attribute$,1) = "p" THEN
          PRINT firstname$; "'s "; attribute$;
                  " is "; politics$(position)
```

Note the use of LEFT$ in this program . This string function
is covered in detail in Chapter 8. It is used here to
examine the first letter of a string. The use of the
function in this context means that the program will ignore
spelling mistakes in the typing of 'attribute$' (but not
'firstname$'), providing the first character is not
misspelt.

Exercises

1 Write a program that will input a list of names. The
  program is then to input a character and print out all
  these names from the list that begin with a specified
  character.

2 A tune can be stored in two parallel one-dimensional
  arrays 'pitch' and 'length' in which successive pairs of
  entries represent the notes of the tune in order. A
  variable 'numberofnotes' can be used to indicate how many
  notes there are in the tune. Write a program that can be
  used to help compose a tune. Commands that the program
  might recognise are:

    "P" for Play the tune (and PRINT the arrays),
    "N" for Next note (INPUT pitch and length),
    "C" for Change note (INPUT note number, pitch and length),
    "Q" for Quit.

3 Write a program that accepts as input four nouns, four verbs and four articles or demonstratives. The program should print ten random sentences. FOR example given input:

    boy, girl, cat, dog
    loves, eats, hits, bites
    a, the, this, that

the output might be

    this dog bites that girl
    this girl eats this dog
    the cat loves a girl
    the boy eats the dog

6.5 Sorting lists into order

Sorting items in a list into some kind of order is a much used technique especially in data processing applications. We may want to sort a list of names into alphabetic order or to sort a list of reference numbers into increasing or decreasing numeric order.

Suppose we have a list 'refno', of unsorted reference numbers that we want to sort into increasing numerical order. If there are n of these numbers, then we should proceed as follows:

    Find the array element with the smallest number
    Swap it with refno(1)

    Find the array element with the second smallest number
    (we need only now look from refno(2) onwards)
    Swap it with refno(2)

                    .

                    .
                  etc.

in other words:

    FOR i = 1 TO n - 1
        Find the element with the ith. smallest number
        Swap it with refno(i)
    NEXT i

Filling in the details, the complete sort becomes:

```
110     FOR i = 1 TO n - 1
120        posnsmallest = i
130        FOR j = i + 1 TO n
140           IF refno(j) < refno(posnsmallest)
              THEN posnsmallest = j
150        NEXT j

160     REM **** now swap ****
170        temp = refno(i)
180        refno(i) = refno(posnsmallest)
190        refno(posnsmallest) = temp
200     NEXT i
```

This particular algorithm is called an exchange sort. Another common but perhaps less obvious technique is called 'bubble sorting'.

Exercises

1 Develop a complete program along the above lines to input a list of names and sort these into alphabetic order.

2 Can you generalise the numeric sorting program such that it will sort the numbers into either increasing or decreasing order according to a control inserted as input at the end of the list of numbers?

3 Write a program to input unsorted information into two parallel arrays. One array is to contain a list of names, the other a list of examination marks. A name and associated mark are of course in corresponding elements in each array. The program is to output a list of names and marks in decreasing mark order. Remember that when the position of a mark is changed, the position of the associated name must also be changed.

6.6 Two-dimensional arrays

Many techniques in science, engineering and commerce deal with data that is organised in two dimensions. Pictures from interplanetary explorers, for example, are enhanced and analysed by computer. The pictures are represented inside the computer as a two-dimensional table of numbers. Each number corresponds to the brightness of a picture element or point. A picture is converted into a table of numbers by a special input device, and after it is processed it is converted back into a picture by a special output device. It is much easier and more natural for a programmer to think in terms of a two-dimensional set of picture elements – the picture retaining its two-dimensional form when referred to in the program – than it would be if the picture elements

were strung out row-wise or column-wise into a one-dimensional array or list.

Consider another example - a 10x8 table of numbers (10 columns, 8 rows), where each number represents the population of the corresponding zone of a 10x8 square mile map. We can retain the two-dimensional nature of the data by storing it in a two-dimensional array 'popmap'. First of all let us see how we tell the computer we are going to use such a structure.

    DIM popmap(10, 8)

This declaration has set up a two-dimensional structure into which we can place the data. We can picture a two-dimensional array as a collection of variables, all of the same type, organised into columns and rows.

1st subscript ⟶
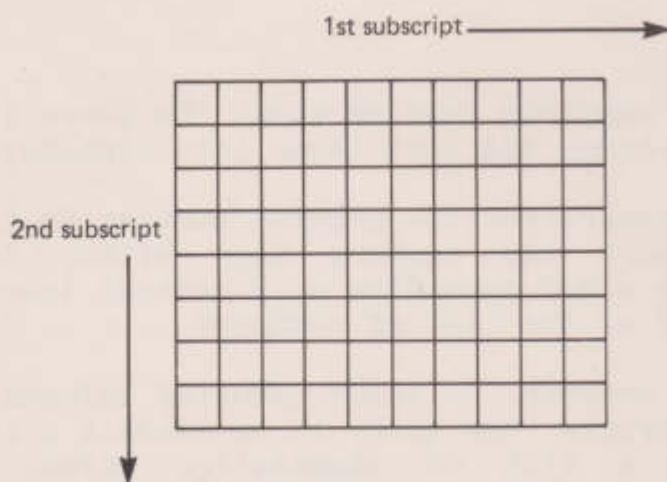
2nd subscript

The first subscript determines a column and the second subscript determines a row. For example 'popmap(3,7)' refers to the location in the 3rd column and the 7th row of the array. Thus we picture the declared array as corresponding in size and shape to to the table of data that we are considering. Actually the two subscripts will run from zero upwards - the computer sets aside space for row zero and column zero. However, it is convenient to ignore row zero and column zero just as we ignored element zero in one-dimensional arrays.

An interesting point to note is that we could equally well visualise the array the other way round, ie. with the first subscript representing a row number and the second a column. In many mathematical applications involving matrix algebra, this alternative picture would correspond to mathematical conventions. It does not matter how we picture our two-dimensional arrays as long as we are consistent.

On the BBC computer, we are used to using x-y coordinates to describe positions on the screen and it will be less confusing if we picture our two-dimensional arrays as having

the same structure as the screen.

To do something systematically with all the locations of a two-dimensional array, we need to use a nested FOR-statement. For example, if we wish to deal with the array a row at a time we need the outline structure:

FOR row = 1 TO 8

> deal with row

and dealing with a row involves a further loop:

FOR column = 1 TO 10

> deal with the location popmap(column, row)

The next example illustrates this. The program inputs the population table described above, stores it in a suitable array, calculates the total population and prints a copy of the population table. The program would, in practice, input from a file, but we omit the extra details to keep things simple.

```
10      DIM popmap(10, 8)
20      FOR row = 1 TO 8
30        FOR column = 1 TO 10
40          INPUT popmap(column, row)
50        NEXT column
60      NEXT row

70      total = 0
80      FOR row = 1 TO 8
90          FOR column = 1 TO 10
100          total = total + popmap(column, row)
110         NEXT column
120     NEXT row
130     PRINT "Total population is "; total

135     @% = &0404
140     PRINT "Populations of individual zones are:"
150     FOR row = 1 TO 8
160       FOR column = 1 TO 10
170         PRINT popmap(column, row);
180       NEXT column
185       PRINT      :REM start a new line for next row.
190     NEXT row
195     @% = &0A0A
```

The formatting statement at line 135 has the effect of dividing the screen into 4-character fields so that 10 numbers will fit on one line (see Appendix 6). The first nested FOR loop causes the data to be read into the two-dimensional array 'popmap', the second accesses the array, performing the required calculation, and the third causes the contents of the array to be printed. When the input for this program is being typed, the 10 numbers in the first row of the table are typed first and these are stored in the first row of the array. Then the 10 numbers in the second row are typed, and so on. Incidentally, the processes of reading the data and accumulating the total could have been carried out simultaneously, the first two nested loops being merged into a single nested loop. This could not be easily done in the next program.

The next program inputs a population table. In addition it inputs four numbers defining a sub-region on the map over which the population is to be added. For example, input of 3, 5, 2, 6 specifies the region lying between columns 3 and 5 and between rows 2 and 6:
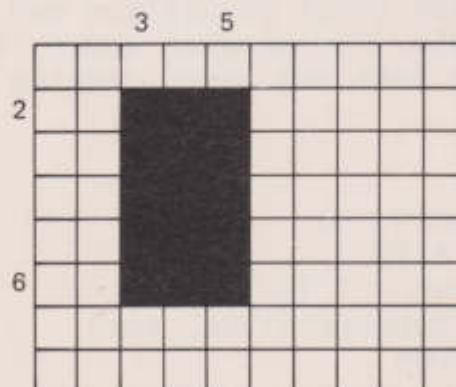

```
10      DIM popmap(10, 8)
20      FOR row = 1 TO 8
30        FOR col = 1 TO 10
40          INPUT popmap(col, row)
50        NEXT col
60      NEXT row

70      subtotal = 0
80      INPUT cola, colb, rowa, rowb
90      FOR row = rowa TO rowb
100       FOR col = cola TO colb
110         subtotal = subtotal + popmap(col, row)
120       NEXT col
130     NEXT row

140     PRINT "Population of sub-zone is "; subtotal
```

The next program reads in the population table and finds every zone with a population less than half the average of its north, south, east and west neighbours.

```
10      DIM popmap(10, 8)
20      FOR row = 1 TO 8
30        FOR col = 1 TO 10
40          INPUT popmap(col, row)
50        NEXT col
60      NEXT row

70      FOR row = 2 TO 7
80        FOR col = 2 TO 9
90          average = (popmap(col,row-1)
                      +popmap(col,row+1)
                      +popmap(col-1,row)
                      +popmap(col+1,row))/4
100         IF popmap(col,row) < average/2 THEN
                PRINT "Region "; col; ","; row;
                                " is underpopulated."
110       NEXT col
120     NEXT row
```

The next program illustrates random access to a two-dimensional array.

A student's week is divided up into 5 days, each of 6 periods numbered 1 to 6. He must attend 20 lectures during each week. The program inputs a list of his lecture times and places. Each lecture time is represented by two integers giving the day (1 to 5) and the period (1 to 6) and these two integers are followed by the number of the room (a positive integer) in which the lecture takes place.

The program prints a timetable for the student in the form:

		Mon	Tue	Wed	Thu	Fri
	1	3	6		5	3
	2	1	6	2	2	2
period	3	2	5	1	5	1
	4	9	7			3
	5		6		8	
	6			7		

where the entry for each period represents the room number in which he should be. We assume that the input data is such that there are no timetable clashes.

```
10      DIM place(5, 6)
20      REM  *** mark all periods as free ***
30      FOR day = 1 TO 5
40       FOR period = 1 TO 6
50          place(day, period) = 0
60        NEXT period
70      NEXT day

80      REM ** fill appropriate periods with room number **
90      FOR lecture = 1 TO 20
100       INPUT day, period, room
120       place(day, period) = room
130     NEXT lecture

140     REM ** print timetable **
150     PRINT "               Mon  Tue  Wed  Thu  Fri"
160     PRINT "               ------------------------"
170     @% = &0505           :REM   see Appendix 6
180     FOR period = 1 TO 6
190        IF period = 3 THEN PRINT "period  ";
                           ELSE PRINT "          ";
200        PRINT ; period;
210        FOR day = 1 TO 5
220           IF place(day, period) = 0 THEN PRINT "      ";
              ELSE PRINT place(day, period);
230        NEXT day
240        PRINT
250     NEXT period
260     @% = &0A0A
```

Exercises

1 You may prefer to have your timetable displayed in the form

                            period

	1	2	3	4	5	6
Mon	3	1	2	9		
Tue	6	6	5	7	6	
Wed		2	1			
Thu	5	2	5		8	7
Fri	3	2	1	3		

Rearrange the second half of the last program so that it does this. You will need to reverse the order of nesting of the FOR loops, and use an ON-GOTO statement to print the name of a day of the week.

2 In a warehouse system, car parts are stored in a two-dimensional stack of pigeonholes, thirty columns long and

ten rows high. This unit is accessed by an automatic electromechanical fetch unit that is given a pair of coordinates by a computer when a part is to be fetched. The values of the elements of a two-dimensional array are to represent the number of items in the pigeonholes and this array should be initialised from a file.

Write a program that repeatedly:

(a) accepts a reference number in the range 1 to 300 and outputs the corresponding pair of coordinates (successive reference numbers are to correspond to consecutive pigeon holes in a row-wise manner starting at the top left pigeonhole);

(b) decrements the appropriate array location by one, thus keeping an up-to-date stock record;

(c) writes an appropriate message if the stock in a pigeonhole falls below ten items.

The input reference numbers will be terminated by two negative numbers. At this point the program should print a two-dimensional representation of the stack, printing a '-' for the pigeonholes in which the stock has fallen below ten items and printing a '+' for the others.

Chapter 7 **Procedures and functions**

A procedure is a section of program to which a name has been given. The programmer can then write the name of the procedure wherever he wants that section of program to be obeyed. This has two main advantages.

Firstly, if the named operation has to be carried out at several different places in a large program, we avoid writing out the same section of program in full at each place.

Secondly, and just as important, careful use of procedures can make a large program easier to write and easier for other people to read. For example, in a payroll program we might require a sequence of operations such as

    add normal hours pay
    add overtime hours pay
    deduct income tax.

Defining a BASIC procedure for each operation allows us to write the sequence of BASIC statements:

    10   PROCaddnormalhourspay
    20   PROCaddovertimehourspay
    30   PROCdeductincometax

where the details of each named operation are defined elsewhere. This makes it much easier to understand what the program is doing than it would be if the details for each calculation were simply written out in full.

7.1 Introductory example

To demonstrate how to define and use a BASIC procedure, we shall write a very simple program that accepts input of a greeting and a name and displays a personalised greeting in the centre of the screen. Of course this program could be written without procedures (in fact it was set as an exercise in Chapter 1), but we shall use it as an introductory example.

In our program, we shall define two separate procedures, one to organise the input and the other to display the greeting. The first two lines of the program will tell the computer what we want done by simply writing the names of

110

the two procedures that are to be obeyed:

```
10      PROCgetinput
20      PROCdisplaygreeting
```

The details of how to 'getinput' must of course be described. So must the details of how to 'displaygreeting'. The complete program is as follows:

```
10      PROCgetinput
20      PROCdisplaygreeting
30      END

100     DEF PROCgetinput
110       INPUT "Message:" message$
120       INPUT "Name of recipient:" name$
130     ENDPROC

200     DEF PROCdisplaygreeting
210       CLS
220       PRINT TAB(5,9);   "***************************"
230       PRINT TAB(10,11); message$
240       PRINT TAB(14,13); name$
250       PRINT TAB(5,15);  "***************************"
260     ENDPROC
```

When the program is obeyed, the first statement encountered is at line 10. This statement is known as a 'procedure call' and tells the computer to go and obey the procedure with the name PROCgetinput. The definition of this procedure lies between lines 100 and 130. When the computer is told to obey a procedure, it finds the procedure heading (line 100 in this case) and obeys the subsequent instructions. When it encounters an ENDPROC statement, it returns to the statement that told it to obey the procedure and carries on from there. In this case, when the computer has obeyed PROCgetinput, it returns and carries on from line 20. This statement tells it to obey the statements that make up the definition of PROCdisplaygreeting (lines 200 to 260). Once this has been done, control returns to the line that called the procedure. Line 30 then tells the computer to do nothing further. The END statement at line 30 is needed to stop the computer from attempting to obey PROCgetinput again. The large gaps in the line numbering between lines 35 and 100 are not strictly necessary, but leaving such gaps makes it easier to extend a program without having to renumber it.

The same procedures could have been called as often as we liked from the main part of the program. For example, let us change the program so that it leaves a greeting displayed until someone hits a key, upon which it accepts more input and displays a new greeting. We replace lines 10 to 30 with:

```
10    PROCgetinput
20    PROCdisplaygreeting
30    key = GET
40    PROCgetinput
50    PROCdisplaygreeting
60    END
```

The rest of the program remains as before. PROCgetinput is now used twice in the main part of the program, but the details are written only once in the procedure definition. The same applies to PROCdisplaygreeting.

Another possibility would be to replace lines 10 to 60 with the following:

```
10    REPEAT
20      PROCgetinput
30      PROCdisplaygreeting
40      key$ = GET$
50    UNTIL key$ = "Q"
55
60    END
```

The procedure definitions again remain as before. This version of the program repeats the process of input and display every time a key is pressed, stopping when someone types Q.

7.2 Procedures and stepwise refinement

In Chapter 5, we presented informally the well-known programming method called 'stepwise refinement'. The first step in writing a complex program should be to sketch an outline of what the program is going to do without getting bogged down in the detailed BASIC instructions required. Using procedures for the logically distinct operations in a program allows us to write our initial outline in BASIC where we invent procedure names to describe operations that we have not yet programmed in detail. Only when we have a clear idea of what each named procedure is to do and how it fits into the overall program do we go on to define each procedure in detail.

In a complicated program, writing one of the procedures may itself be a difficult programming task and a procedure can itself be defined in terms of other procedures.

We illustrate this approach to programming by writing a program that plays a simple guessing game. The computer makes up a random number between 1 and 100 and the user seated at the keyboard must attempt to guess the number. For each wrong guess, the computer tells the user whether the guess was too high or too low and the process terminates

when the number is guessed correctly. The program finishes by analysing and commenting on the player's performance.

We start by outlining the process that the program will carry out:

```
10    PROCinitialise
20    REPEAT
30      PROCtryguess
40    UNTIL gotit
45
50    PROCanalyseperformance
60    END
```

PROCinitialise will set the variables required by the program to their starting values. Each call of PROCtryguess will input the player's next guess and test it against the secret number (invented by PROCinitialise). PROCanalyseperformance will report how many attempts were needed to guess the number and will print an appropriate message.

PROCinitialise and PROCtryguess are defined as:

```
100    DEF PROCinitialise
110       secretnumber = RND(100)
120       tries = 0
130       gotit = FALSE
140    ENDPROC

200    DEF PROCtryguess
210    LOCAL guess
220       tries = tries + 1
230       INPUT "Guess:" guess
240       IF guess = secretnumber THEN
               gotit = TRUE : ENDPROC
250       IF guess < secretnumber THEN
                PRINT "Too small!";
           ELSE  PRINT "Too big!";
260       PRINT " Try again.";
270    ENDPROC
```

In PROCtryguess, we have specified at line 210 that the variable 'guess' is 'local' to the procedure. This variable is available for use only while PROCtryguess is being obeyed. Variables declared at the start of a procedure in this way can not be used after ENDPROC has been obeyed. It is recommended that any variable which is used only within a particular procedure should be declared locally to that procedure. The computer will then ensure that the programmer does not accidentally use the same variable for conflicting

purposes in different parts of a large program. If a variable with the same name is used elsewhere in the program, its value will automatically be held in a different storage location, thus eliminating any possibility of conflict.

When PROCtryguess is obeyed, the first test that it makes at line 240 is to decide whether the player has guessed the secret number. If he has, then 'gotit' is set to true and an ENDPROC statement is obeyed immediately. This terminates the procedure and lines 250 to 270 are ignored. If the player has not guessed correctly, the ENDPROC at line 240 is ignored and the computer carries on to obey lines 250 and 260 before encountering the ENDPROC at line 270. The extra ENDPROC at line 240 has been used to create the effect of a complex IF-THEN-ELSE statement where the 'ELSE part' occupies more than one numbered line.

Finally, we define PROCanalyseperformance:

```
300    DEF PROCanalyseperformance
310      PRINT "That took "; tries; " attempts."
320      IF tries <= 7 THEN  PRINT "Very good!"
         ELSE PRINT "You need to improve your strategy."
330    ENDPROC
```

As your programs become more complicated, you should try to adopt the approach illustrated above. Any logically separate section of program should be defined as a procedure and given a descriptive name. In practice, writing a lot of procedures of only two or three lines, like PROCinitialise, might be considered excessive. In such cases, we leave it to you to find a compromise that suits you.

7.3 Menu selection

In this section we return to the fragments of a stock control program that were presented in Section 6.4. We now present a program in which the the various stock control operations are defined as procedures and a menu selection procedure is used to determine which operation is to be carried out next.

```
10    DIM refno(100),name$(100),stock(100),priceof(100)
20    PROCinitialise
30    PROCprintmenu
40    REPEAT
50      INPUT command$
60      PROCmenuselect
70    UNTIL command$ = "F"
80    PROCcreatenewstockfile
90    END
```

```
100     DEF PROCinitialise
          *** defined in Appendix 2 ***
180     ENDPROC

200     DEF PROCcreatenewstockfile
          *** defined in Appendix 2 ***
280     ENDPROC

300     DEF PROCprintmenu
310       PRINT:PRINT
320       PRINT "Type M for menu"
330       PRINT "Type S for stock enquiry"
340       PRINT "Type P for price enquiry"
350       PRINT "Type O to process order"
360       PRINT "Type F to finish and file stock list"
370       PRINT:PRINT
380     ENDPROC

400     DEF PROCmenuselect
410       IF command$ = "M" THEN PROCprintmenu
420       IF command$ = "S" THEN PROCprintitemsinclass
430       IF command$ = "P" THEN PROCfindprice
440       IF command$ = "O" THEN PROCprocessorder
450     ENDPROC

500     DEF PROCprintitemsinclass
510       INPUT "Name of item", class$
520       FOR i = 1 TO sizeofstock
530         IF name$(i) = class$ THEN
                PRINT "refno "; refno(i); " (";
                       name$(i);") "; stock(i); " in stock"
540       NEXT i
550     ENDPROC

600     DEF PROCfindprice
610       INPUT "Stock number", number
620       i = 0
630       REPEAT
640         i = i + 1
650       UNTIL number = refno(i)
660       PRINT "Price of ref. no "; refno(i); " (";
                     name$(i); ") is "; priceof(i)
670     ENDPROC
```

```
700     DEF PROCprocessorder
710     INPUT "Stockno", number, "Quantity", quantity
720        i = 0
730        REPEAT
740           i = i + 1
750        UNTIL number = refno(i)
760        IF stock(i)-quantity<0 THEN
              PRINT "not enough in stock"
           ELSE   PRINT "Cost: "; quantity * priceof(i) :
                 stock(i) = stock(i) - quantity
770     ENDPROC
```

There are, of course, many more options that could be added
to this program. These are left as exercises.

7.4 Data validation

When it is possible to check or validate the information
input to a program this should always be done. Not only does
this guard against the possibility of a run-time error, the
cause of which might not be immediately apparent to the
user, but worse - invalid data may cause erroneous results
which go unnoticed. The use of procedures can make it easier
to incorporate such tests in the design of a program. In
simple cases, we can use a structure such as:

INPUT information

IF information is ok THEN

> obey main part of program
> (defined as a procedure)

ELSE   PRINT error message

The next program uses this structure. It calculates the cost
of a holiday by multiplying the duration in days by a daily
rate that is seasonally variable. The input consists of two
dates each in the form day, month. We assume that the months
will be either identical or consecutive. Also, we assume
that it is the second month which determines the seasonal
rate. We have somewhat simplified the checks for the
purposes of this demonstration - not every month has 31 days
and we have not catered for numbers <=0.

The use of the procedure PROCholidaycost not only makes
clear the structure of the program. It also overcomes the
restriction on the number of statements that we can have
between THEN and ELSE. It is also convenient to define a
separate procedure for working out the duration of the
holiday.

```
10    lowrate = 57
20    midrate = 72
30    peakrate= 87
40    INPUT day1, month1, day2, month2
50    IF day1 <= 31 AND day2 <= 31 AND month2 <= 12 AND
         ((month1 = month2 AND day1 < day2)
             OR (month2 = month1+1))
      THEN PROCholidaycost
      ELSE PRINT "You have typed erroneous dates."
60    END

100   DEF PROCholidaycost
110     PROCworkoutduration
120     ON month2 GOTO 121, 121, 122, 122, 122, 123,
                       123, 123, 123, 122, 121, 121
121       cost = duration * lowrate  : GOTO 130
122       cost = duration * midrate  : GOTO 130
123       cost = duration * peakrate
130     PRINT "Cost of holiday is "; cost
140   ENDPROC

200   DEF PROCworkoutduration
210     IF month1 = month2 THEN
              duration = day2 - day1 + 1 : ENDPROC
220     ON month1 GOTO 223, 221, 223, 222, 223, 222,
                       223, 223, 222, 223, 222, 223
221       tillendofmonth = 28 - day1 : GOTO 230
222       tillendofmonth = 30 - day1 : GOTO 230
223       tillendofmonth = 31 - day1
230     duration = tillendofmonth + 1 + day2
240   ENDPROC
```

In the above program, PROCholidaycost is obeyed only if the input is acceptable (within the limits of the test used). If the input is erroneous, the user will have to RUN the program again. If a program is required to automatically request more input until a correct set of values are typed, then the following structure can be used:

```
REPEAT
    INPUT information
    inputok = result of test on input
    IF NOT inputok THEN
        PRINT "Error! please retype information."
UNTIL inputok
```

```
obey main part of program
```

In the case of the holiday program, use of this structure would give:

```
10      lowrate = 57
20      midrate = 72
30      peakrate= 87

40      REPEAT
50        INPUT day1, month1, day2, month2
60        inputok = day1<=31 AND day2<=31 AND month2<=12 AND
                    ((month1=month2 AND day1<day2)
                     OR (month2=month1+1))
70        IF NOT inputok THEN
              PRINT "Erroneous dates. Please retype."
80      UNTIL inputok
85
90      PROCholidaycost
95      END
```

Exercises

1 A company's employees are paid £4.85 per hour for a standard 35 hour week. Overtime hours during the week are paid at 1.25 times this basic rate and overtime hours at the weekend are paid at 1.5 times this rate. Income tax is paid at the rate of 30% on the first 100 pounds of a week's pay, 40% on the next 100 pounds and 50% on the remainder. Write a program that accepts, as input, the number of weekday overtime hours worked by an employee in one week. The program should display a payslip giving details of his normal pay, overtime pay and deductions for that week. The main part of the program should consist of the three procedure calls presented in the introduction to this chapter.

2 Rewrite the computer-assisted learning program presented at the end of Chapter 5, but use a procedure so that the main part of the program becomes:

```
REPEAT
   PROCsetproblem
   INPUT "More? Y/N:" reply$
UNTIL reply$ = "N"
```

3 Rewrite some of the other programs of Chapter 5 using procedures, as in Exercise 1 above, to make clear the logical structure of the program.

4 In order to run the stock control program for the first time, a stock file will have to be set up. Write a separate program to do this.

5 Extend the stock control program to include an 'Update stock level' command. After typing this command, the user will type a reference number and a quantity that is to be added to the previous stock level for that item.

6 Extend the stock control program to include a 'Check stock level' command. This command should list all those items for which the number in stock is less than 10.

7 Extend the holiday booking program to include complete data validation. One neat way of doing this is to set up an array of 12 locations, one for each month. Each location should contain the number of days in the month. The number of days in February will depend on the year which will have to be supplied as additional input. The values in this array can then be used in testing the validity of a day.

7.5 Parameters

We have seen how simple procedures can be used to enable a program to carry out the same operation at different points in a program. A common requirement is for similar, but not necessarily identical, operations to be carried out at different points in a program.

For example, in Chapter 2, we wrote a program that worked out how many coins of each denomination were needed to make up a given amount of change. The operation used to calculate the number of 50p pieces to be included in the change was very similar to the operation used for each of the other coin denominations. We can reprogram this process using a procedure as follows:

```
10    INPUT "Change", change

20    PROChowmany(50) :    PROChowmany(20)
30    PROChowmany(10) :    PROChowmany( 5)
40    PROChowmany( 2) :    PROChowmany( 1)
```

The intention here is that PROChowmany is the name of a procedure that is going to be used six times. Each time the procedure is called, it is supplied with a 'parameter' in brackets telling it which denomination of coin to deal with next. The complete program is as follows:

```
10    INPUT "Change", change

20    PROChowmany(50) :   PROChowmany(20)
30    PROChowmany(10) :   PROChowmany( 5)
40    PROChowmany( 2) :   PROChowmany( 1)
50    END

100   DEF PROChowmany(denomination)
110   LOCAL noofcoins
120     noofcoins = change DIV denomination
130     change = change MOD denomination
140     PRINT "No of "; denomination; "s "; noofcoins
150   ENDPROC
```

In the procedure definition, the operation to be carried out
by the procedure is specified in terms of 'denomination',
which is a parameter. When the procedure is called, this
parameter is given an actual value which is to be used when
the procedure is obeyed. Thus, obeying the statement

    PROChowmany(50)

causes the procedure definition to be obeyed with:

    denomination = 50

Similarly, obeying the statement

    PROChowmany(20)

causes the procedure to be obeyed with:

    denomination = 20

A parameter supplied in brackets when the procedure is
called is usually referred to as an 'actual parameter'.
A procedure with parameters will perform some general
task defined in terms of these parameters. When the
parameters are given actual values, the procedure performs a
particular version of the general task. Such a facility is
extremely important in modern programming.
The next procedure is defined in terms of two parameters.

```
100   DEF PROCtabulate(x, n)
110   LOCAL i
120     PRINT "Table of multiples of "; x  :   PRINT
130     FOR i = 1 TO n
140       PRINT i * x
150     NEXT i
160     PRINT
170   ENDPROC
```

When this procedure is called, multiples of the number supplied as its first parameter are printed, the number of multiples being specified as a value for the second parameter. If we call this procedure as follows:

```
10    PROCtabulate(3.72, 4)
```

'x' is given the value 3.72, 'n' is given the value 4 and the procedure definition is obeyed. Output is therefore:

Table of multiples of 3.72

```
        3.72
        7.44
       11.16
       14.88
```

The actual parameters supplied to this procedure can in fact be any two expressions. For example, the fragment:

```
10    r = 4.3 : noofmultiples = 5
20    PROCtabulate(r, noofmultiples)
30    PROCtabulate(SQR(SQR(r)), 2*noofmultiples)
40    END
```

will tabulate the first 5 multiples of 4.3 and the first 10 multiples of the fourth root of 4.3.
The fragment:

```
10    DIM a(4)
20    FOR i = 1 TO 4 : INPUT a(i) : NEXT i

30    FOR i = 1 TO 4
40       PROCtabulate(a(i), 10)
50    NEXT i
60    END
```

will call the procedure 4 times and output 4 tables, one for each entry in the array 'a'.

In other programming languages, it is usually possible to pass information out of a procedure by changing the value of one of its parameters while the procedure is being obeyed. In BBC BASIC, parameters can only be used for passing information into a procedure and these parameters are sometimes known as input parameters. If information calculated in a procedure is to be used outside that procedure, the information must be placed in a global variable - any variable that is not a parameter or a local variable. For example, in the program at the beginning of

this section the global variable 'change' is altered by each call of the procedure. In fact this global variable is used both to transfer information into and out of the procedure.

As we mentioned earlier, a procedure definition can include calls of other procedures. The next program reads two positive integers and prints them in descending order. The integers are to be output on separate lines and, when an integer is output, the millions are to be followed by a comma and the thousands are to be followed by a comma. Thus, instead of printing 5674251 the program should print 5,671,251

```
10      INPUT first, second
20      IF first > second THEN
            PROCcommaprint(first) : PROCcommaprint(second)
        ELSE
            PROCcommaprint(second) : PROCcommaprint(first)
30      END

100      DEF PROCcommaprint(n)
110      LOCAL millions, thousands, units
120        millions = n DIV 1000000
130        thousands = (n MOD 1000000) DIV 1000
140        units = n MOD 1000
150        IF millions > 0 THEN
               PRINT ;millions; ","; :
               PROCzeroprint(thousands) : PRINT ","; :
               PROCzeroprint(units) : PRINT :
           ENDPROC
160        IF thousands > 0 THEN
                  PRINT ;thousands; ","; :
                  PROCzeroprint(units) : PRINT :
           ENDPROC
170        PRINT ;units
180      ENDPROC

200      DEF PROCzeroprint(m)
210        IF m < 100 THEN PRINT "0";
220        IF m < 10  THEN PRINT "0";
230        PRINT ;m;   :REM  See Appendix 6
240      ENDPROC
```

Writing the main part of this program (lines 10 to 30) is straightforward, provided that we avoid getting bogged down in the problems involved in printing a number with the commas inserted. Designing PROCcommaprint can then be viewed as a programming problem which is slightly easier than the original problem. The need for PROCzeroprint arises because, for example, a number of thousands has to be written with a full three digits in the context of a number like 2,067,003. In the number 67,003 the thousands are printed with no extra

zeros, and a normal PRINT statement is used.

7.6 String parameters

The parameter of a procedure can be a string. For example, the following program displays a message surrounded by stars in the centre of the screen, pauses for 5 seconds, displays a second message, and clears the screen after a further 10 seconds.

```
 10     PROCdisplay("Merry", "Christmas")
 20     PROCtimedelay(500)
 30     PROCdisplay("Happy", "New  Year")
 40     PROCtimedelay(1000)
 50     CLS : END

100     DEF PROCdisplay(line1$, line2$)
110     LOCAL r
120       CLS
130       PRINT TAB(13,9);   "**************"
140       FOR r = 10 TO 14
150         PRINT TAB(13,r); "*"; TAB(25,r); "*"
160       NEXT r
170       PRINT TAB(13,15); "**************"

180       PRINT TAB(17,11); line1$
190       PRINT TAB(15,13); line2$
200     ENDPROC

300     DEF PROCtimedelay(d)
310     LOCAL stoptime
320       stoptime = TIME + d
330       REPEAT : UNTIL TIME > stoptime
340     ENDPROC
```

PROCdisplayboard is defined in terms of two parameters that are both strings. The first time it is called (at line 10), we have

```
line1$ = "Merry"
line2$ = "Christmas"
```

The second time it is called (at line 30), we have

```
line1$ = "Happy"
line2$ = "New  Year"
```

PROCdisplay could also have been supplied with string variables as its actual parameters. For example, we could write:

```
10    INPUT "Message, part1:" part1$
20    INPUT "         part2:" part2$
30    PROCdisplay(part1$, part2$)
40    END
```

In the above program, PROCtimedelay has been defined in such a way that the BASIC variable TIME is left unchanged. This mechanism will be useful in applications where a time delay is required, but where TIME is also being used for other purposes.

Procedures can have any combination of numeric and string parameters. In the next program, we have extended PROCdisplay so that it requires two additional parameters telling it where on the screen to display the message. It is used to display three messages on different parts of the screen simultaneously.

```
10    CLS
20    PROCdisplay(5,  5, "Merry", "Christmas")
30    PROCdisplay(20, 7, "Happy", "New  Year")
40    PROCdisplay(10,15, "From",  "BBC Micro")
50    END

100   DEF PROCdisplay(col, row, line1$, line2$)
110   LOCAL r
120     PRINT TAB(col,row); "**************"
130     FOR r = row + 1 TO row + 5
140       PRINT TAB(col, r); "*"; TAB(col+12, r); "*"
150     NEXT r
160     PRINT TAB(col, row + 6); "**************"
170     PRINT TAB(col + 4, row + 2); line1$
180     PRINT TAB(col + 2, row + 4); line2$
190   ENDPROC
```

Exercises

1 Write a program that accepts three sums of money, each
   input as a real number of pounds. The program should
   tabulate the three sums of money in two columns, one
   column for the pounds and one for the pence. The program
   should add up the three sums of money and print the total
   at the bottom of the two columns. Use a procedure,
   PROCprintmoney, that takes a real number of pounds as its
   parameter and prints it as two integers, pounds and
   pence.

2 Write a program that draws a selection of circles with
   different centres and different radii. The program should
   use a procedure:

   PROCcircle(centrex, centrey, radius)

   (A program for drawing a single circle appeared in
   Chapter 4.)

3 Write a program that plays a simple tune and then plays
   it again in a different key. Use a procedure:

   PROCplaytune(key)

   where the parameter indicates the pitch of the keynote.

4 Write a program that displays several Christmas trees,
   each outlined in stars and containing the message
   "Merry Christmas" followed by someone's name. Use a
   procedure to print a Christmas tree in a specified
   position and containing a specified name:

   PROCchristmas(row, column, name$)

7.7 Functions

If the result of some process is a single value, then a
function is sometimes an elegant alternative to a procedure.
We have already seen how to use the standard functions SQR,
SIN, COS and so on and we will now look at how to write our
own functions.

First let us look at the ways in which a function differs
from a procedure. Certainly, they are both separate modules
of program text referred to by name, but they differ in the
way in which they are called. Functions are called by using
them in arithmetic expressions - that is the first
difference. The second difference is that the result of
obeying the function is a single value which replaces the
function call in the originating expression. Let us
illustrate this by considering:

```
y = x + SQR(2)
```

When this statement is being obeyed, the computer obeys the
definition of the standard function SQR, and a number - the
result of obeying the function - replaces the subexpression
SQR(2). In the case of a standard function like SQR, the
definition of the function is already stored as part of the
BASIC system, but it is also possible for the programmer to
define his own functions. In BBC BASIC, a programmer defines
a function in a way that is very similar to the way in which
procedures are defined. A function defined in this way can
be used in exactly the same way as the standard functions.

This program reads 3 pairs of numbers and adds the larger
of the first pair, the larger of the second pair and the
larger of the third pair. A function is used to find the
larger of two numbers.

```
10    INPUT a,b, p,q, x,y
20    PRINT FNmax(a,b) + FNmax(p,q) + FNmax(x,y)
30    END

40    DEF FNmax(first,second)
50       IF first > second THEN  = first
         ELSE  = second
```

The effect of calling a function is the calculation of a
single result. Since calling a function produces a single
result, we must indicate, somewhere in the   function
definition, what this result is to be. Instead of ENDPROC,
the function terminates when a statement of the form:

```
= expression
```

is obeyed. The value of this expression is returned as the
value of the subexpression used to call the function. Apart
from this need to return a value as its result, the
definition of the function can include as many statements as
are necessary to calculate this value.

When the above program is obeyed, evaluation of the
subexpression 'FNmax(a,b)' causes the function definition to
be obeyed with 'first' set to the value of 'a' and 'second'
set to the value of 'b'. If the function is called when we
have the situation

```
a = 4.79
b = 5.64
```

then the function definition is obeyed with

```
first = 4.79
second = 5.64
```

and the statement

    = second

is obeyed as a result of obeying the IF statement. The value of the subexpression 'FNmax(a,b)' is therefore 5.64 and this is the value used in subsequent evaluation of the larger expression:

    FNmax(a,b) + FNmax(p,q) + FNmax(x,y)

If the main part of the program had included a statement such as

    PRINT FNmax(63.45, 61.23)

the function would have been obeyed with

    first = 63.45
    second = 61.23

and the value returned as the result of the function in this case would be 63.45, this value being displayed by the PRINT statement. The actual parameters can of course be any expressions that will have numeric values when the program is obeyed. This means that the actual parameters in a call of the function can themselves involve further function calls. We have already seen how we can do this with the standard functions:

    z = SQR(SQR(x) + SQR(y))
    PRINT SQR(SQR(SQR(z)))

FNmax can be used in the same way:

    PRINT FNmax(SQR(2), SQR(3))

will print

1.73

and

    PRINT FNmax(FNmax(6.2, 7.4), FNmax(2.3, 9.5))

will print

9.5

In the last case, the function calls are evaluated as follows:

$$\text{FNmax}(\ \underbrace{\text{FNmax}(6.2,\ 7.4)}_{7.4},\ \underbrace{\text{FNmax}(2.3,\ 9.5)}_{9.5}\ )$$

$$9.5$$

As another example, the last program could have been written (rather longwindedly) as:

```
10      INPUT a,b, p,q, x,y
20      PRINT FNsumof(FNmax(a,b), FNmax(p,q), FNmax(x,y))
30      END

40      DEF FNmax(first,second)
50          IF first > second THEN  = first
            ELSE  = second

60      DEF FNsumof(n1, n2, n3)
70      = n1 + n2 + n3
```

Exercises

1 Write a program that inputs two pairs of marks and outputs the average of the first pair of marks followed by the average of the second pair of marks. The program should use a function, FNaverage to calculate the average of two marks.

2 A motor rally takes place on one day and, for each car, a start time and a finish time are recorded. Each car also has a handicap time which is to be subtracted from the true time taken for the rally in order to determine an adjusted time. Write a program which reads the start time, finish time and handicap time for one car and which prints the true time and the adjusted time for that car. Each time is input as two integers (hours and minutes) separated by a space. The program should do all its calculations in minutes, the conversion from hours and minutes being done by a function, FNinputtime, that can be used by, for example:

```
start = FNinputtime
```

The conversion back to hours and minutes should be done by a procedure PROCprinttime which takes a parameter representing a time in minutes and prints it in hours and

minutes.

7.8 A final example - a noughts and crosses program

In this section, as a final example on the use of procedures
and functions, we shall develop a program that is much more
complicated than any that we have written so far. We shall
write a program that plays a game of 'noughts and crosses'
(tic-tac-toe) with an opponent seated at the keyboard. The
program will make extensive use of procedures and functions,
partly to assist us in the program design process and partly
to save us from writing similar pieces of program in several
different places. The main part of the program is as
follows:

```
10    PROCstartgame
20    PROCdisplayboard

30    REPEAT
40       IF computersturn THEN PROCcomputer
                          ELSE PROCopponent
50       PROCdisplayboard
60       PROCtestgameover
70    UNTIL gameover
80    PROCannouncewinner
90    END
```

In this outline, we have not yet committed ourselves to any
details about the game. Indeed, we have not yet committed
ourselves to the game of noughts and crosses. The same
outline structure could be used for any board game such as
draughts (checkers) or chess. However, the correct logical
structure for playing a board game is established, breaking
the task into logically distinct modules, each one of which
is considerably easier to cope with than the whole problem.

All the details of how to start the game off by setting
variables to their initial values and deciding who starts
will appear in PROCstartgame. Before we can define this
procedure, we must decide on how to represent the 'board'.
Each square on the noughts and crosses board can contain an
"X", an "O" or a blank. This suggests using an array, each
location of which contains a character. We could use a 3 x 3
array:

    DIM board$(3,3)

and select a square by using two numbers specifying a column
and a row. However, we prefer to use a one-dimensional
array:

```
DIM board$(9)
```

and we can then select a square by using a single number where the squares are numbered as follows

```
1 2 3
4 5 6
7 8 9
```

Thus at some stage in the game, we might have

board$(1)	"O"
board$(2)	"."
board$(3)	"X"
board$(4)	"."
board$(5)	"X"
board$(6)	"."
board$(7)	"O"
board$(8)	"."
board$(9)	"."

stored as

```
O   X
  X
O
```

where "." represents a blank square.

PROCstartgame is defined as:

```
100     DEF PROCstartgame
110     LOCAL square, reply$
120       DIM board$(9)
130       movesmade = 0
140       gameover = FALSE
150       blank$ = "."
160       FOR square = 1 TO 9
170         board$(square) = blank$
180       NEXT square

190       INPUT "Do you want to start, Y/N", reply$
200       IF reply$="Y" THEN
                computersturn = FALSE :
                opp$  = "X" : comp$ = "O"
          ELSE computersturn = TRUE  :
                comp$ = "X" : opp$  = "O"
210     ENDPROC
```

A variable 'movesmade' will be used for counting the moves so that we can easily recognise when the board is full.

A variable 'blank$' contains the character that is used to mark a blank square. Giving a name to this character will

make it easier to change it to something else if we wish.

The player who makes the first move is "X" and the other player is "O". The variable 'comp$' contains the character ("X" or "O") being played by the computer and 'opp$' contains the character being played by the opponent.

We shall keep PROCdisplay as simple as possible. Each time that it is called, the characters in the noughts and crosses board will be printed on the screen:

```
300     DEF PROCdisplayboard
310     LOCAL square
320       PRINT : PRINT
330       FOR square = 1 TO 9
340         PRINT board$(square); " ";
350         IF square MOD 3 = 0 THEN PRINT
360       NEXT square
370       PRINT : PRINT
380     ENDPROC
```

The display produced by this procedure could be considerably improved but this is left as an exercise.

We must now define the two move making procedures. For the time being, the computer will select its moves completely at random. A random number in the range 1 to 9 is repeatedly generated until the number of a blank square is obtained. The character 'comp$' ("X" or "O") is then inserted in that square, the move is counted and 'computersturn' is set to FALSE. Note that when we want to improve the computers strategy we know that it is this procedure that we have to alter.

```
500     DEF PROCcomputer
510     LOCAL move
520       REPEAT
530         move = RND(9)
540       UNTIL board$(move) = blank$
550       PRINT "I play in square "; move
560       board$(move) = comp$
570       movesmade = movesmade + 1
580       computersturn = FALSE
590     ENDPROC
```

In PROCopponent, the move number made is obtained by using INPUT:

```
700    DEF PROCopponent
710    LOCAL move
720      INPUT "Your move, square number:" move
730      board$(move) = opp$
740      movesmade = movesmade + 1
750      computersturn = TRUE
760    ENDPROC
```

The procedure PROCtestgameover involves testing for rows of "X"s and rows of "O"s and it is convenient to define it in terms of other procedures:

```
900    DEF PROCtestgameover
910      PROCtestcomputerwin
920      IF computerwin THEN gameover = TRUE : ENDPROC
930      PROCtestopponentwin
940      IF opponentwin THEN gameover = TRUE : ENDPROC
950      IF movesmade = 9 THEN gameover = TRUE
960    ENDPROC
```

Of course we should not call both 'PROCtestcomputerwin' and 'PROCtestopponentwin'. The logical elaboration required to prevent both procedures being called is left as an exercise (below). PROCtestcomputerwin will search the board for a group of three of the computer's character in a straight line configuration. PROCtestopponentwin will search for a straight line configuration of the opponent's character. It is convenient to define these procedures in terms of a common function:

```
1000    DEF PROCtestcomputerwin
1010      IF FNrowof(comp$) THEN    computerwin = TRUE
                           ELSE    computerwin = FALSE
1020    ENDPROC

1100    DEF PROCtestopponentwin
1110      IF FNrowof(opp$) THEN    opponentwin = TRUE
                          ELSE    opponentwin = FALSE
1120    ENDPROC
```

FNrowof is a logical function that produces a result TRUE or FALSE. It is given a character as its parameter and tests all straight line combinations of three squares to see if there is such a combination of squares that all contain the given character. Each combination of three squares is tested by calling another function with three parameters indicating the numbers of the three squares to be tested:

```
1200    DEF FNrowof(letter$)
1210      IF FNthree(1,2,3) THEN   = TRUE
1220      IF FNthree(4,5,6) THEN   = TRUE
1230      IF FNthree(7,8,9) THEN   = TRUE
1240      IF FNthree(1,4,7) THEN   = TRUE
1250      IF FNthree(2,5,8) THEN   = TRUE
1260      IF FNthree(3,6,9) THEN   = TRUE
1270      IF FNthree(1,5,9) THEN   = TRUE
1280      IF FNthree(3,5,7) THEN   = TRUE
1290    =FALSE
1295
1300    DEF FNthree(s1, s2, s3)
1310    =  (board$(s1) = letter$ AND board$(s2) = letter$
                              AND board$(s3) = letter$)
```

Note that the equals sign is being used for two different purposes on line 1310. The first one indicates that the value of the logical expression that follows (TRUE or FALSE) is the result of the function. The others are relational operators testing for equality.

There are many other ways in which the tests carried out by FNrowof could be arranged, by using various combinations of REPEAT loops. However, in view of the small number of combinations to be tested, the above approach is certainly the most straightforward.

The definition of PROCannouncewinner completes the program:

```
1400    DEF PROCannouncewinner
1410      IF computerwin THEN PRINT "I win!"
          ELSE IF opponentwin THEN PRINT "You win!"
              ELSE PRINT "It's a draw."
1420    ENDPROC
```

Several of the following exercises involve making improvements to the above program. You should notice when doing these exercises that the way in which the program was written makes it very easy to identify the part of the program text that needs to be changed to implement a particular improvement.

Exercises

1 The game 'last one loses' is played as follows:
    Two players take turns at removing at least one and not more than three counters from a stack that initially contains 'n' counters where 'n' is chosen at random at the start of each game. The player who is forced to remove the last counter loses.
  Write a program to play a game of 'last one loses'. Use

the same approach to writing this program as was used in writing the noughts and crosses program.

2 Extend the noughts and crosses program so that it starts by telling the opponent how to type moves. Use a procedure PROCinstructions.

3 As it stands, PROCopponent allows the computer's opponent to make illegal moves. Improve this procedure so that this possibility is dealt with.

4 If the computer has just made a move in the noughts and crosses program, it is a waste of time for PROCtestgameover to call PROCtestopponentwin. Similarly, if the opponent has just made a move, PROCtestcomputerwin should not be called. Reorganise PROCtestgameover so that unnecessary tests are eliminated.

5 The display produced by the noughts and crosses program could be improved in various ways. The code numbers of the various squares could be permanently displayed in the corner of the screen. The board itself could be permanently displayed in the centre of the screen and only the relevant square in the board changed (using TAB) when a move is made. Make these improvements.

6 Rewrite the noughts and crosses program so that it uses a two-dimensional array:

    DIM board$(3,3)

to represent the board. Which version of the program do you prefer?

7 Try and improve PROCcomputer so that the program plays more 'intelligently'. The procedures PROCtestcomputerwin and PROCtestopponentwin might be useful here. For example, the program could try each possible move in turn and use PROCtestcomputerwin to see if it was a winning move.

Chapter 8 Special effects with characters and strings

In this chapter, we discuss various special effects that can be obtained with characters and strings on your BBC micro, but first, we present a little more information on how characters are stored and processed inside the machine.

8.1 How characters are stored

A character is stored inside the computer as an integer that occupies 8 bits or one byte. There is an internationally agreed standard set of codes for the commonly used characters. These are the ASCII codes (American Standard Code for Information Interchange).

The first table in Appendix 7 contains a list of the normal display characters and their ASCII codes. Actually, in MODE 7, a different international set of characters is used, the Teletext characters. This means that, in MODE 7, a few of the characters displayed on the screen are different from the picture on their keys. However, these characters are used fairly infrequently.

Conversion functions: ASC and CHR$

Two special functions are available for converting the first character of a string into its numeric code (ASC) and for converting a numeric code into a single character string (CHR$). The statement

    PRINT ASC("+"), ASC("A"), ASC("a")

will print the numbers

          43   65   97

whereas the statement

    PRINT CHR$(38); CHR$(75); CHR$(122)

will print the characters

    &Kz

The following program inputs a sequence of 10 ASCII codes and builds up a string containing the 10 corresponding

characters.

```
10    chars$ = ""
20    FOR i = 1 TO 10
30      INPUT "Next code ", code
40      chars$ = chars$ + CHR$(code)
50    NEXT i
60    PRINT "These codes make up "; chars$
```

The VDU statement

The VDU statement provides an alternative way of sending characters to the display hardware. The word VDU is followed by a list of character codes separated by commas and these codes are sent one by one to the screen. If the codes represent visible characters, then these characters will appear on the screen. Thus the two statements

    VDU 65, 66, 67, 88, 89, 90

and

    PRINT "ABCXYZ";

have exactly the same effect. However, the VDU statement is normally used for sending invisible characters or special control codes to the display hardware. For example, the ASCII codes from 1 to 31 are reserved for special purposes on the BBC computer and if one of these codes is sent to the display hardware, it is intercepted and handled specially.

For example, 8 is the code for 'backspace' and the statement

    VDU 8, 8, 8

will move the cursor back 3 character positions on the current line. Note that the same effect can be obtained with

    PRINT CHR$(8); CHR$(8); CHR$(8);

or with

    back3$ = CHR$(8)+CHR$(8)+CHR$(8);
        .
        .
        .
    PRINT back3$;

The string 'back3$' contains three invisible control characters that are sent to the display hardware when the string is printed.

In general there are many different ways of sending a sequence of characters (visible and control) to a device. The most appropriate depends on the application. The last method above would be most convenient in a program that frequently required to send three backspaces. Once the string has been defined and given a name, the name can be included in a PRINT statement wherever it is required.

Various special control codes will be introduced and explained as they are required in this and later chapters. Tables of the various codes and a brief description of their effects appear in Appendix 7.

8.2 Coloured text and its uses

Control of the colour of characters is easily effected on the BBC computer from a BASIC program. The particular facilities discussed in this section are MODE, which selects a particular graphics and text mode for the screen, and COLOUR, which selects particular foreground and background colours for the characters. The modes available to you depend on whether you have a 32K or 16K machine. With the 16K (Model A) machine MODES 7 to 4 are available and with the 32K (Model B) machine MODES 7 to 0 are available. The computer normally operates in MODE 7 and you can always return to this mode by pressing BREAK or by typing MODE 7. For example you may be developing a program that runs in MODE 2. After an erroneous run, a listing in MODE 2 may not be particularly readable, and you could type MODE 7 (without a line number) and LIST the program in MODE 7. Alternatively you could add the following two lines at the bottom of every program:

```
300    keypress = GET
310    MODE 7
```

After a program in say MODE 2 has run pressing any key will clear the screen and return to MODE 7. You can also switch to MODE 7 by typing CONTROL-V followed by 7 or you might prefer to define one of the user-defined function keys to switch to mode 7 and list the program. (See Appendix 1 to see how to do this.)

A summary of the character facilities and the colours normally available in each mode follows. You can see from this that, in general, as the number of characters available on the screen increases the colour options decrease. This is a point we will be examining in much more detail when we deal with graphics. Mode 7 uses the Teletext standard display characters which cannot be changed by the user.

mode	colours available	characters per line	lines
0	2	80	32
1	4	40	32
2	16	20	32
3	2	80	25
4	2	40	32
5	4	20	32
6	2	40	25
7	Teletext display	40	25

Colour codes for 2-colour modes (0,3,4,6)

foreground	background	colour
0	128	black
1	129	white

Colour codes for 4-colour modes (1 and 5)

foregroumd	background	colour
0	128	black
1	129	red
2	130	yellow
3	131	white

Colour codes for 16-colour mode (MODE 2)

foreground	background	colour
0	128	black
1	129	red
2	130	green
3	131	yellow
4	132	blue
5	133	magenta
6	134	cyan
7	135	white
8	136	flashing black-white
9	137	flashing red-cyan
10	138	flashing green-magenta
11	139	flashing yellow-blue
12	140	flashing blue-yellow
13	141	flashing magenta-green
14	142	flashing cyan-red
15	143	flashing white-black

To select a particular mode we write, for example,

    10   MODE 2

To select a particular foreground colour we write, for example,

    20   COLOUR 1

To select a particular background colour we write, for example,

    30   COLOUR 129

You can see from the tables that a background colour colour code is 128 plus the foreground colour code. Another way of putting it is to say that COLOUR refers to the foreground if the code stated is less than or equal to 15 and to the background if the code is greater than or equal to 128. The COLOUR statement changes the colour (foreground or background) used for subsequent printing (but not for drawing). The foreground and background colours of information already on the screen do not change. When a character is printed, the character itself is displayed in the current foreground colour against a small rectangle of the current background colour. If the background colour has not been changed, the background round the character is indistinguishable from the existing background colour on the screen. When the background colour is changed, characters are subsequently printed with a local background that is different from that round the characters previously printed. This behaviour is demonstrated below.

The following example uses local background colours. It prints two messages on the same line using a different combination of background and foreground colours for each message.

    10   MODE 5
    20   COLOUR 1
    30   COLOUR 130
    40   PRINT " red on yellow ";
    50   COLOUR 2
    60   COLOUR 129
    70   PRINT " yellow on red "

Incidentally, note that if you type LIST after running this program the computer will continue to operate in the same mode using the most recently set up foreground and background colours. You should use one of the methods mentioned earlier to return to MODE 7.

If you want a background colour for the whole screen then

140

CLS (clear screen) can be used. This, as the name implies, clears the screen entirely and sets the whole screen to the current background colour. Note the difference in effect between:

```
10    MODE 5
20    COLOUR 1
30    COLOUR 130
40    CLS
50    PRINT "red on yellow"
```

and:

```
10    MODE 5
20    COLOUR 1
30    COLOUR 130
40    PRINT "red on yellow"
```

Further examine the effect of:

```
10    MODE 5
20    COLOUR 1
30    COLOUR 130
40    CLS
50    PRINT "red on yellow"
60    COLOUR 2
70    COLOUR 129
80    PRINT "yellow on red on yellow"
```

The next program uses two isolated background colours as a simple method of plotting a bar graph or chart. It displays a bar chart representing the values in the DATA statement. These values must each be in the range 0 to 19. The bars of the chart are drawn horizontally.

```
10    DATA 3,5,8,9,10,15,19,12,8,4
20    MODE 5
30    FOR i = 1 TO 5
40      COLOUR 129
50      READ h
60      FOR j = 1 TO h
70        PRINT" "; : NEXT j
80      PRINT
90      COLOUR 130
100     READ h
110     FOR j = 1 TO h
120       PRINT" "; : NEXT j
130     PRINT
140   NEXT i
```

In this program the local background colour alternates as
blue and yellow. We print spaces to make up the bars. A
space consists entirely of background colour and the effect
of printing a space without invoking CLS is to print a
rectangle of the current background colour against a black
whole screen background.

8.3 The use of flashing colour

If you have a Model A which does not support MODE 2, you
should still read this section and then move on to Section
8.4 which explains how to accomplish similar effects in MODE
5. Flashing colour available in MODE 2 can be used in
computer dialogue to accentuate a particular part of a
message or a piece of text. In the computer-assisted
learning program of Chapter 5, we were posing questions
like:

    16 + 25 = ?

Suppose we want to set up the screen as follows for this
question:
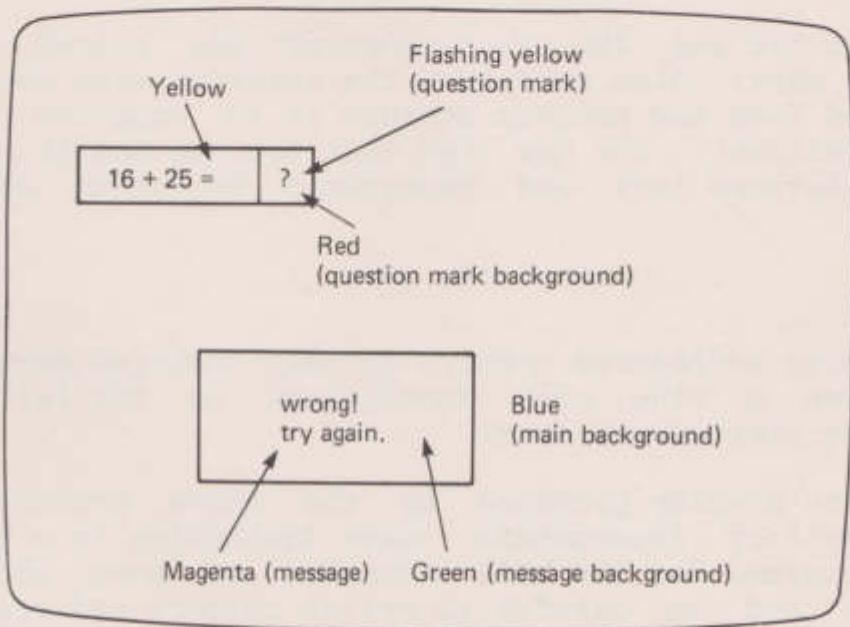
This could be achieved by using the following procedures in
MODE 2 (almost! see Exercise 1 below):

```
200   DEF PROCdisplayquestion
210     COLOUR 3
220     COLOUR 132
230     CLS
240     FOR i = 1 TO 4 : PRINT : NEXT i
250     PRINT "      "; a; " + "; b; " = ";
260   ENDPROC

300   DEF PROCgetanswer
310     COLOUR 11
320     COLOUR 129
330     INPUT answer
340   ENDPROC

400   DEF PROCwronganswer
410     FOR i = 1 TO 4 : PRINT : NEXT i
420     COLOUR 5
430     COLOUR 130
440     PRINT "                "
450     PRINT "    wrong!      "
460     PRINT " try again. "
470     PRINT "                "
480   ENDPROC
```

Note lines 310 and 320 that control the colour of the question mark. Also note that the question mark should not be printed from the program because it is supplied by the INPUT statement. You may find that you can obtain a better contrast between text and background by using different colours.

Exercises

1   The above procedures require further detailed development to give a blue area immediately to the left of the message display. Do this.

2   Once the display produced by the above procedures is satisfactory incorporate these facilities in a complete CAL program. Use the TAB function to move about the screen and be careful to switch colours and erase text (by overprinting with spaces) where appropriate. You may find that you need time delays to give the user time to read messages before they are erased.

3   Further develop the bar graph program so that the bars are two lines thick.

4   Write a bar graph program that produces a more conventional display with the bars running vertically. This is quite difficult - it can be done with TAB statements, but you may find it easier to build up the

histogram in a two-dimensional array storing different codes for different coloured 'spaces'. The whole array can then be printed in the appropriate order, switching colours as required.

8.4 Extending the standard colour range of the Model A

In this section we introduce a facility that allows the two or four colours available in MODE 0, 1, 3, 4, 5 or 6 to be any two or four of the 16 colours available in MODE 2. This means, for example, that flashing colours can be used with the Model A. It also means that 16 colours can be used in a program run on a Model A, with the restriction that only four can appear on the screen at any instant.

Recall that to specify a colour for subsequent display the COLOUR statement is used with a numeric parameter. The colour selected by this code number can be changed by the VDU 19 statement. This redefines or transforms the colour definitions. For example:

```
10    MODE 5
20    COLOUR 1
30    PRINT "colour 1 is red"
```

has the effect that you have already seen. The effect of

```
10    MODE 5
20    VDU 19, 1, 9, 0,0,0
30    COLOUR 1
40    PRINT "colour 1 is now flashing red"
```

is to change the definition of COLOUR 1 from its original red to flashing red (colour number 9 in the list for MODE 2). It is still called COLOUR 1 in the program.

The general form is:

VDU 19, | colour code number | , | actual colour number | , 0,0,0

This is a particular use of the VDU statement to send one of the control codes (19 in this case) mentioned in the first section of this chapter. Whenever the code 19 is sent, it must be followed by 5 other codes, the last three of which are usually 0. (If you miss out the zeros, the next three characters sent to the screen by PRINT or VDU statements are ignored.)

If we now refer back to the computer-assisted learning program, we can see that a similar effect in the question text can be achieved in MODE 5 as follows:

```
200   DEF PROCdisplayquestion
210     VDU 19, 3, 4, 0,0,0
220     COLOUR 2
230     COLOUR 131
240     CLS
250     FOR i = 1 TO 4 : PRINT : NEXT i
260     PRINT "    "; a; " + "; b; " = ";·
270   ENDPROC

400   DEF PROCwronganswer
410     FOR i = 1 TO 4 : PRINT : NEXT i
420     VDU 19, 0, 5, 0,0,0
430     VDU 19, 1, 2, 0,0,0
440     COLOUR 0
450     COLOUR 129
460     PRINT "              "
470     PRINT "   wrong!     "
480     PRINT " try again. "
490     PRINT "              "
500   ENDPROC
```

The VDU 19 statements could, in fact, be obeyed only once at the start of the program. Remember in MODE 5 only 4 different colours can appear on the screen at once and for this reason we have omitted the colour change for the question mark and its background.

The complete set of 4 static colours could be changed to 4 flashing colours by using 4 VDU statements as follows:

```
VDU 19, 0,  9, 0,0,0
VDU 19, 1, 10, 0,0,0
VDU 19, 2, 11, 0,0,0
VDU 19, 3, 12, 0,0,0
```

The process of colour redefinition is best imagined schematically as follows: suppose we want to make the colour codes 0, 1, 2, 3 refer to actual colours 2, 6, 12, 4 using the statements:

```
VDU 19, 0,  2, 0,0,0
VDU 19, 1,  6, 0,0,0
VDU 19, 2, 12, 0,0,0
VDU 19, 3,  4, 0,0,0
```

These statements set up 4 pathways to 4 of the possible actual colours. With the VDU statement we can set up any 4 pathways to 4 of the 16 possible actual colours.

actual colour
numbers

```
COLOUR parameters        0
  in MODE 5              1
                         2
                         3
                         4        screen colours
                         5
     0                   6        green
     1                   7        blue
     2                   8        cyan
     3                   9        flashing blue
                        10
                        11
                        12
                        13
                        14
                        15
```
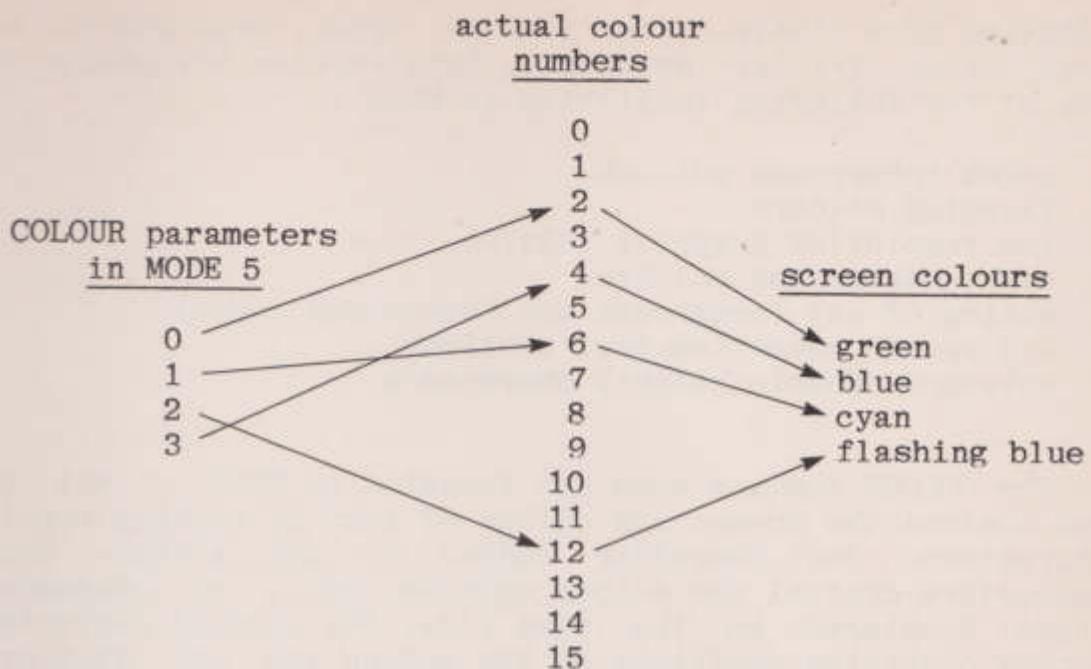
When the VDU statement is used to change actual colours in this way, material already displayed on the screen is also affected. The following program illustrates this.

```
10    MODE 5
20    PRINT "Characters in COLOUR 3"

30    FOR actual = 1 TO 15
40        keypress = GET
50        VDU 19, 3, actual, 0, 0, 0
60    NEXT actual
```

If the above fragment is RUN, then each time a key is pressed the actual colour associated with code number 3 is changed and the colour of the message previously displayed with colour code 3 also changes. It is this feature that we exploit later in the chapter on animation.

8.5 Economical colour – extending MODE 7

The memory requirements for MODE 7 down to MODE 0 are 1K, 8K, 10K, 10K, 16K, 20K, 20K and 20K respectively. This memory is needed during program execution and reduces the memory available for program and data. It would clearly be advantageous if the colour facilities could be extended into MODE 7 which only requires 1K. For example in a chess playing program that requires a lot of memory, it would be advantageous to program the display in MODE 7. A chess display does not suffer too much from the lack of fine graphic detail. This can be achieved at the expense of the

addition of a little tedium to the BASIC programming and some other trivial drawbacks. This section introduces the use of the following facilities in MODE 7:

    seven foreground colours
    flashing colours
    low resolution graphics (80x75)
    eight background colours
    mixing of any foreground and background colour
    hiding and revealing text portions
    enlarged (double height) characters

The COLOUR command does not function in MODE 7 but you can control the format and colour of text by sending special characters (or character codes) to the screen. These characters control the colour options, etc., of subsequent items displayed on the same line. The control characters occupy character positions on the screen and are displayed as spaces. They only control the display for the remainder of the line in which they are introduced. Many effects need a pair of special characters, one to turn the effect on and the other to turn it off. Thus programming a display in MODE 7 needs a different approach in detail.

Coloured text in MODE 7
The colour of text in a PRINT statement is controlled by the insertion of colour control characters. Thus:

```
10 MODE 7
20 PRINT [red control] ;"red"; [blue control] ;"blue"; [green control] ;"green"
```

will print:

    red blue green

with each word displayed in its implied colour if the appropriate controls are included as indicated. As mentioned above the control characters themselves are displayed as spaces. The colour control codes are as follows:

control codes	foreground colour
129	red
130	green
131	yellow
132	blue
133	magenta
134	cyan
135	white

The above PRINT statement is therefore:

```
20      PRINT CHR$(129); "red"; CHR$(132); "blue";
                CHR$(130); "green"
```

and the colour effects only last for this one line. Any subsequent output lines revert to normal MODE 7 facilities unless they also contain colour control characters. This approach is rather more tedious than the COLOUR command which of course controls all subsequent output. Another tedious aspect of this facility is that displays must be carefully planned to take into account the appearance of one or two more spaces being printed due to the control characters. This is the price we pay for the economical use of computer memory.

The control characters can be incorporated in a string just like any other character, and this facility allows us to set up a mechanism for emphasising the occurrence of a particular word or phrase in a piece of text. Examine the effect of the following:

```
10      MODE 7
20      red$ = CHR$(129) + "red" + CHR$(135)
30      blue$= CHR$(132) + "blue"+ CHR$(135)
40      PRINT "Roses are"; red$;
                "Violets are"; blue$;
                "Sugar is sweet"
```

The string 'red$' will contain 5 characters: the word red preceded by the red control character and succeeded by the white control character.

Coloured background in MODE 7
Background colour control in MODE 7 is somewhat tedious, requiring 4 control characters for conventional use. It can, however, be used to effect when hidden text is to be revealed, the background strip being selectively removed to reveal the 'hidden' text 'underneath'.

The control characters used are as follows:

control code	effect
156	Reset background colour to black
157	Set background colour to current foreground colour

In the previous section the 'roses are red' ditty was displayed with the words "red" and "blue" in red and blue foreground colours. The next fragment displays the verse with the words "red" and "blue" as white letters on a red and blue background.

```
10    MODE 7
20    red$ = CHR$(129) + CHR$(157) + CHR$(135)
              + "red" + CHR$(156)
30    blue$= CHR$(132) + CHR$(157) + CHR$(135)
              + "blue" + CHR$(156)
                    .
                    .
```

You can see from this that 3 controls are required to set up a contrasting background:

(1)   a control to set a current foreground colour,
(2)   a control to make the background this colour,
(3)   a control to reset the foreground to another colour.

If the 3rd control is omitted then the foreground and background colours are the same and the text is 'hidden'. This effect is exploited in the next program which forms a skeleton on which a guessing game could be built. A white strip appears at the top of the screen hiding a film title. Hitting any key will cause one more word to be revealed.

```
10    MODE 7
20    DIM onoff$(10)
30    on$ = CHR$(156) : off$ = CHR$(157)
40    FOR i = 1 TO 6 : onoff$(i) = off$ : NEXT i

50    FOR i = 1 TO 7
60       PRINT TAB(1,1);
70       PRINT onoff$(1); "In";
80       PRINT onoff$(2); "the";
90       PRINT onoff$(3); "heat";
100      PRINT onoff$(4); "of";
110      PRINT onoff$(5); "the";
120      PRINT onoff$(6); "night";
130      continue = GET
140      onoff$(i) = on$
150   NEXT i
```

In practice of course many film titles would have to be stored in string arrays and randomly selected. If the film title were stored in an array, then lines 70 to 120 could be replaced by a loop (or, even better, a procedure containing a loop). We will not be concerned here with such complications. In the above program initially the background colour is set to white so that the text is hidden. As the main loop progresses the control character codes in the spaces between words are reset from 157 (set background colour to current foreground) to 156 (reset background colour to black). The white 'mask' hiding the text shrinks

to the right revealing one word at a time.

Note that a similar effect could be obtained, but without the white strip, by using CHR$(152) as the 'off' character and a space as the 'on' character. This would not be quite as effective as the change of background colour used above.

Exercises

1 Alter the last program so that the user selects a particular word to be revealed. E.g. typing 5 should reveal only the 5th. word.

Large characters in MODE 7

Double height letters are available in MODE 7 but not in any other mode (unless you design them yourself – see Chapter 11). Again the effect is obtained by using control characters, and these are:

control code	effect
140	single height characters
141	double height characters

This pair of control characters can be seen as 'turn on' (141) and 'turn off' (140) signals for the large character option. Again control only lasts for one line and the provision of a 'turn on' and 'turn off' control means that large and small letters can be mixed on the same line.

To print text in large letters two PRINT statements are required (the text occupies two lines). The controls must appear in both PRINT statements as must the text. Examine the effect of the following:

```
10    MODE 7
20    PRINT CHR$(141); "BIG"; CHR$(140); "small";
          CHR$(141); "BIG"
30    PRINT CHR$(141); "BIG"; CHR$(140); "small";
          CHR$(141); "BIG"
```

which mixes large and standard height characters. The first PRINT statement displays the upper half of the text and the second PRINT statement the lower half.

Different colours can be introduced on different lines or halves of the characters. Examine the effect of the following:

```
10    MODE 7
20    PRINT CHR$(129); CHR$(141);"BIG BIG BIG"
30    PRINT CHR$(131); CHR$(141);"BIG BIG BIG"
```

This should produce the text with the upper half coloured red and the lower coloured yellow. Note that the texts in the two consecutive PRINT statements must be 'aligned'.

Misalignment is used in the next program to produce a special effect. The upper half of the text "BIG BIG BIG" coloured red appears initially to the left of centre. The lower half, coloured yellow, appears initially to the right of centre. The two halves then move in to meet each other under the speed control of line 70 which produces the delay between displays.

```
10    MODE 7
20    CLS
30    FOR i = 1 TO 10
40      j = 20 - i
50      PRINT TAB(i,10);CHR$(129);CHR$(141);"BIG BIG BIG";
60      PRINT TAB(j,11);CHR$(131);CHR$(141);"BIG BIG BIG";
           TAB(40)
70      FOR k = 1 TO 100 : NEXT k
80    NEXT i
```

Note the use of the TAB function in this program, particularly TAB(40) in statement 50. You can see why this is necessary by observing the effect of removing it.

Low resolution graphics in MODE 7

The MODE 7 resolution is 40x25 characters. If the special Teletext graphics characters are used (see Appendix 7), then the resolution is 'increased' to 80x75. Each Teletext graphics character is a pattern of colour/no colour on a 2x3 matrix. The snag is of course that the individual 80x75 pixels are not accessible from the program. An (x,y) coordinate in the program is a coordinate in the 40x25 system and this makes for quite tricky programming.

The Teletext graphics character facility must be switched on by using one of the Teletext codes 145 to 151. These codes are listed in Appendix 7. After one of these codes has appeared on a line, many of the characters that are normally displayed as ASCII characters (ASCII codes 32 to 63 and 95 to 126) are displayed as Teletext graphics characters. A table of the ASCII characters that are replaced by graphics characters appears in Appendix 7. In addition, the same range of graphics characters can be obtained, in a slightly more systematic way, by using codes 160 to 191 and 224 to 255. Appendix 7 shows how to calculate the code in this higher range for a particular graphics character.

As an example of the difficulties involved in using the Teletext graphics facilities let us now look at the problem of drawing a rectangle. We could use 3 loops - one for the top edge, one for the sides and one for the bottom edge. However note that if we use characters 44 (horizontal bar) and 53 (vertical bar) to produce say a 30x20 rectangle using:

```
10      MODE 7
20      PRINT CHR$(147);
30      FOR i = 1 TO 30 : PRINT CHR$(44); : NEXT i
40      PRINT

50      FOR i = 1 TO 20
60        PRINT CHR$(147); CHR$(53);
              TAB(30); CHR$(53)
70      NEXT i

80      PRINT CHR$(147);
90      FOR i = 1 TO 30 : PRINT CHR$(44); : NEXT i
100     PRINT
```

we will end up with a rectangle that looks like this:
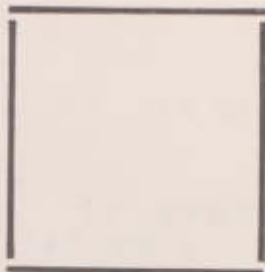
with gaps and mismatches at the corners, for reasons that
are obvious if you examine the 2x3 patterns involved. We
should have used a different graphics character for each
edge.

Incidentally note that in the above program the
colour/graphics control is outside the loop for the
horizontal lines but inside the vertical line loop. This is
of course due to the fact that as in the previous sections
control lasts for only one line.

Defining shapes with Teletext graphics characters

Areas or shapes to be coloured can be handled by using
combinations of teletext characters. This technique is
exactly analogous to the user-defined character facility
available in the other modes. For example the 'man in the
rain' can be made up from the characters whose codes are
shown in the corresponding table. (We have used the higher
range of graphics codes in this example.)
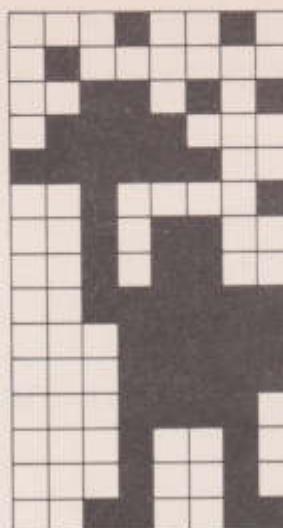
168	242	224	225
174	191	173	224
32	245	255	240
32	234	255	191
32	250	32	245

(Note that in the diagram black is the foreground and white the background. The diagrams in the User Guide are the inverse of this.) This could be printed by the following:

```
10   MODE 7
20   PROCprintmaninrain(1, 1)
30   END

100  DEF PROCprintmaninrain(x, y)
110   PRINT TAB(x,y);        : VDU 146, 168, 242, 224, 225
120   PRINT TAB(x, y + 1); : VDU 146, 174, 191, 173, 224
130   PRINT TAB(x, y + 2); : VDU 146,  32, 245, 255, 240
140   PRINT TAB(x, y + 3); : VDU 146,  32, 234, 255, 191
150   PRINT TAB(x, y + 4); : VDU 146,  32, 250,  32, 245
160   PRINT
170  ENDPROC
```

Simple animation in MODE 7

The above shape could be moved horizontally over the screen by simply calling the print procedure in a FOR loop controlling the value of 'x'.

```
10    MODE 7
20    y = 1
30    FOR x = 0 TO 20
40     PROCprintmaninrain(x, y)
50    NEXT x
60    END
```

The pattern is reprinted in consecutive x positions and appears to be moving because the previous pattern is erased. The erasing is achieved 'automatically' by the printing of the control code 146 as a space. MODE 7 is very fast and you

may want a delay in this loop:

    35   FOR delay = 0 TO 200 : NEXT

Character animation techniques are discussed in detail in Chapter 11. The only special point to bear in mind with Teletext characters is the erasing effect of the unavoidable control codes.
Note that this effect works only when moving in a horizontal direction. Animation in any other direction must be carried out as detailed in Chapter 11, by using an erase and reprint sequence. Character animation in MODE 7 is rather tedious. Character processing in other modes is almost as fast and carries the added bonus of user defined character shapes.

Using ASCII characters to specify graphics shapes

Instead of using the numeric codes for the Teletext graphics characters, a string of the corresponding ASCII characters can be output by a PRINT statement and, provided that the characters are preceded on the line by a graphics control code, they will appear as Teletext graphics characters. Thus, for example, the statements:

    10   PRINT CHR$(145);" p¦";CHR$(255);"¦p"
    20   PRINT CHR$(145);"j7+";CHR$(255);"'k5"
    30   PRINT CHR$(145);"j}¦";CHR$(255);"¦~5"
    40   PRINT CHR$(145);"j7";CHR$(255);"k7";CHR$(255);"!"

will display the following:

CHR$(255) does not have a key and is normally displayed as a white block. Consult the table of graphics characters in Appendix 7 to see how the above program works. Note that the ASCII characters needed for a particular shape will change as the position on the screen at which the shape is printed is changed. You should be able to work out why this happens. You can observe the effect by playing with the program below.

A Teletext graphics sketchpad program

To overcome some of the tedium involved in working out which ASCII characters to include in PRINT statements in order to generate a Teletext graphics picture, we now reproduce a listing of a sketchpad program. The program is a little too complex to be explained here, but you should be able to understand it after you have completed the book. It will enable you, however, to use low resolution graphics more easily. The program is driven like a previous sketchpad program, with N, S, E, and W used as directional control characters. Typing T (for Text) displays the set of ASCII characters needed to make up the sketched pattern. B and F (Background and Foreground) switch the plot on and off. Q (Quit) terminates the program.

```
  5    MODE 7
 10    DATA 1,2,4,8,&10,&40
 20    DATA &FE,&FD,&FB,&F7,&EF,&BF
 30    DIM bit%(5),nobit%(5),scr%(40,25)
 40    PROCsetmasks
 50    graph%=151: text%=32
 60    PROCclear
 70    VDU 23;8202;0;0;0;

 80    REPEAT
 90      VDU 31,x,y
100      b=2*suby+subx
110      IF fore THEN ch%=scr%(x,y) OR bit%(b)
                  ELSE ch%=scr%(x,y) AND nobit%(b)
120      scr%(x,y) = ch%
130      on%=FNcode(ch% OR bit%(b))
140      off%=FNcode(ch% AND nobit%(b))
150      REPEAT
160         VDU on%,8
170         c$=INKEY$(10)
180         VDU off%,8
190         TIME=0
200         REPEAT : UNTIL TIME>10 OR c$<>""
210      UNTIL c$ <>""
220      VDU FNcode(ch%),8
230      IF c$="N" THEN PROCnorth
240      IF c$="S" THEN PROCsouth
250      IF c$="E" THEN PROCeast
260      IF c$="W" THEN PROCwest
270      IF c$="B" THEN fore = FALSE
280      IF c$="F" THEN fore = TRUE
290      IF c$="G" THEN PROCshow(graph%)
300      IF c$="T" THEN PROCshow(text%)
310      IF c$="C" THEN PROCclear
320    UNTIL c$="Q"
330    PRINT TAB(0,24);
340    END
```

```
370    DEF PROCnorth
380      IF suby>0 THEN suby=suby-1
         ELSE IF y>0 THEN y=y-1 : suby=2
390    ENDPROC


410    DEF PROCsouth
420      IF suby<2 THEN suby=suby+1
         ELSE IF y<23 THEN y=y+1 : suby=0
430    ENDPROC


450    DEF PROCeast
460      IF subx=0 THEN subx=1
         ELSE IF x<39 THEN x=x+1 : subx=0
470    ENDPROC


490    DEF PROCwest
500      IF subx=1 THEN subx=0
         ELSE IF x>1 THEN x=x-1 : subx=1
510    ENDPROC


530    DEF PROCshow(m%)
540     LOCAL
550        FOR l=0 TO 24
560          PRINT TAB(0,l);
570           VDU m%
580        NEXT l
590        VDU 31,x,y
600    ENDPROC


630    DEF PROCsetmasks
640      LOCAL i
650        FOR i=0 TO 5
660          READ   bit%(i)
670      NEXT i
680      FOR i= 0 TO 5
690        READ nobit%(i)
700      NEXT i
710    ENDPROC


730    DEF FNcode(patt%)
740      IF patt%=127 THEN=255
750      IF patt%=96  THEN=95
760      IF patt%=35  THEN=96
770    =patt%
```

```
790    DEF PROCclear
800    LOCAL col, row
810      CLS
820      FOR row=0 TO 24
830        VDU 31,0,row, graph%
840        FOR col = 0 TO 39
850          scr%(col,row)=&20
860        NEXT col
870      NEXT row
880      x=1 : y=0 : subx=0 : suby=0
885      fore = FALSE
890    ENDPROC
```

Area fill

Large areas of the screen are most conveniently filled using a nested FOR loop structure and printing Teletext character 255. Note that this however reduces the effective shape resolution back to 40x25. For example, the followings prints a rectangle of selected colour and dimensions at a selected position:

```
100    DEF PROCrectangle(colour, sx, sy, height, width)
110      FOR row = sy TO sy + height
120        PRINT TAB(sx-1, row); CHR$(colour);
130        FOR column = sx TO sx + width
140          VDU 255
150        NEXT column
160        PRINT
170      NEXT row
180      PRINT
190    ENDPROC
```

The PRINT between the FOR statements sets the colour for each line, and the PRINT between NEXTs gives a new line.

Including control codes in PRINT strings

In this final section on the use of Teletext control codes, we mention a useful way of incorporating Teletext control codes into PRINT statements in a program in such a way that the space occupied by the program is minimised. This method has the additional advantage that, when the program is listed in MODE 7, the strings will appear in the listing exactly as they will appear when printed in a program run.

For example, suppose we want to print a message in yellow on a red background and then switch back to white text on a black background. We could do this with

```
10    PRINT CHR$(129); CHR$(157); CHR$(131);
           "yellow on red"; CHR$(135); CHR$(156);
```

or with

```
10    VDU 129, 157, 131 : PRINT "yellow on red"; :
                                          VDU 135, 156
```

An alternative that is very economical in terms of space occupied by the program is to insert the control codes between the quotation marks in the PRINT statement. Although these codes appear as spaces on the screen, they are still stored in the screen memory and once they have been displayed on the screen (looking like spaces) they can be copied into strings that are part of a program.

To do this for the above example, we need to type either of the above lines without the line number so that it is obeyed immediately. The message we want is now displayed on the screen complete with the required control characters. The cursor editing facilities can now be used to copy the (invisible) control codes and the message into a string in a program:

```
10    PRINT "    yellow on red    ";
```
              copied using the cursor
              edit facilities from
              the message displayed
              on the screen

Here is a sample of some output produced using a combination of Teletext facilities together with a photograph of part of the program that produced it. (We have taken some liberties with the program layout for the purpose of photographing it.)

8.6 String functions

String functions are standard functions that operate on a string and return either a number (for example the number of characters in a string) or another string (for example, part of the string being operated on).

String functions: numeric values (INSTR, LEN)

One of the most useful numeric string functions is INSTR. This searches for the occurrence of a substring within a string and returns a number that is the start position of the substring. For example:

```
10    string$ = "catatonic"
20    sub1$ = "cat"
30    sub2$ = "tonic"
40    PRINT INSTR(string$,sub1$), INSTR(string$,sub2$)
```

would print 1 followed by 5. This means that "cat" appears in "catatonic" starting at character position 1 and "tonic" appears starting at character position 5. If no match is found then zero is returned. Upper and lower case alphabetic characters are different in this context. If the total number of occurrences of a substring within a string is to be counted we could proceed as follows:

```
10    quote$ = "curiouser and curiouser cried Alice"
20    INPUT "Word", word$
30    startlooking = 1 : noofocc = 0
40    REPEAT
50      position = INSTR(quote$, word$, startlooking)
60      IF position > 0 THEN noofocc = noofocc + 1:
               startlooking = position + LEN(word$)
70    UNTIL position = 0  OR  startlooking>LEN(quote$)
80    PRINT "Occurrences of "; word$; " are "; noofocc
```

In this program we have used an optional extra argument or parameter in INSTR. This causes the search to begin from the position in the string specified by this parameter. If a match is found the search must restart from a position at the end of the word just found. Otherwise there are no more occurrences of the string and the search can terminate immediately. Another feature introduced in this program is the function LEN. This returns an integer that is the length of the string, or the number of characters in it. The next illustration of INSTR could perhaps be the root of a medical diagnosis program tree. The aim of this fragment is to determine the tense of a reply to the question: "What is the matter with you?". The reply would be typed in by a patient

or even input via a voice recognition system and the program
is to be 'user friendly' (as they say), and interpret a wide
range of possible replies.

```
10      INPUT reply$
20      IF    INSTR(reply$, "have been") <> 0
           OR INSTR(reply$, "have felt") <> 0
           OR INSTR(reply$, "had"       ) <> 0
           OR INSTR(reply$, "was"       ) <> 0
        THEN past = TRUE
        ELSE past = FALSE
30      IF past THEN PRINT"Are you feeling better now?"
        ELSE PRINT "What are your symptoms?"
```

It would, of course, have to be both more comprehensive and
more flexible, but the fragment imparts a flavour of the
dialogue processing that can be carried out using INSTR.
(Incidentally see the User Guide, p281, for an error in the
INSTR implementation in the first version of the operating
system [1982]).

String functions: string values (LEFT$, RIGHT$, MID$)
The functions LEN and INSTR return numeric values. There are
three string functions that return fragments or substrings
of the string that they are operating on. These are LEFT$,
RIGHT$ and MID$.

    substring$ = LEFT$(string$, n)

sets 'substring$' to the first n characters of 'string$'

    substring$ = RIGHT$(string$, n)

sets 'substring$' to the last n characters of 'string$'.

    substring$ = MID$(string$, m, n)

sets 'substring$' to the fragment of 'string$' which
contains n characters starting at character number m. The
next program illustrates the use of MID$, and tests a
sentence to see if it is a palindrome. For example, MADAM IM
ADAM is a palindrome. If spaces are ignored it reads the
same backwards. (The other palindromic sentences that make
up this (in)famous dialogue are not fit for publication!)
The program also uses most of the string handling techniques
introduced so far, such as comparing and appending strings.
We will assume for the purpose of the next program that the
palindrome is typed in upper case letters throughout.

```
10      INPUT sentence$
20      PROCtakeoutspaces
30      FOR i = LEN(newsentence$) TO 1 STEP -1
50        backwards$ = backwards$ + MID$(newsentence$, i, 1)
60      NEXT i
70      IF backwards$ = newsentence$ THEN
             PRINT "sentence is a palindrome"
        ELSE PRINT "sentence is not a palindrome"
80      END

90      DEF PROCtakeoutspaces
100       newsentence$ = ""
110       FOR i = 1 TO LEN(sentence$)
120         letter$ = MID$(sentence$, i, 1)
130         IF letter$ <> " " THEN
                 newsentence$ = newsentence$ + letter$
150       NEXT i
160     ENDPROC
```

Note that the fragment extracted by MID$ is one character long. This is an extremely common use of MID$. Note also the procedure to extract spaces. Processing strings to manipulate spacing and change words forms the basis of word-processing programs. Right justification of text means that inter-word gaps have to be adjusted, such that the last character of a particular word in a line falls exactly on the end of the line. Changing every word for another involves searching (INSTR), for the word to be changed, making the necessary substitution, together with any length adjustment, that this substitution causes. The next program looks at this problem. It inputs a sentence, then a pair of words. The first word is the word to be replaced, the second the replacement word.

```
10      INPUT sent$
20      INPUT oldword$, newword$
30      startlooking = 1
40      REPEAT
50        postn = INSTR(sent$, oldword$, startlooking)
60        IF postn <> 0 THEN PROCinsertnewword:
                 startlooking = startlooking + LEN(newword$)
70      UNTIL postn = 0
80      PRINT sent$
90      END

100     DEF PROCinsertnewword
110       sent$ = LEFT$(sent$, postn - 1) + newword$
            + RIGHT$(sent$, LEN(sent$)-postn-LEN(oldword$)+1)
120     ENDPROC
```

String functions : conversion utilities (STR$, VAL)
Two associated functions for conversion between strings and
numbers are STR$ and VAL. STR$ converts a numeric constant
or the value of a numeric variable into a string constant.
The following fragment sets up a string in 'date$', namely
"19 January"

```
10    day = 19
20    month$ = "January"
30    date$ = STR$(day) + " " + month$
```

The function VAL does the opposite. It converts a numeric
string into a numeric constant. The following would place
the numeric constant 19 in numeric variable 'day'.

```
10    date$ = "19th. January"
20    day  = VAL(LEFT$(date$, 2))
```

The next program is a less trivial illustration of VAL. The
program sorts three records containing a student name and
mark into descending order of mark. We could handle the
names and marks as separate entities, the names as strings
and the marks as numeric variables, but this lengthens the
program considerably and makes the programmer responsible
for always remembering the link between the mark and the
associated name. We can sort the strings as single entities
by proceeding as we did elsewhere for sorting strings into
alphabetic order except that this time we use the numeric
part of the string to control the sort. 'record1$',
'record2$' and 'record3$' are each 14 characters long, say,
with each mark contained in the last 4 characters.

```
10    INPUT record1$, record2$, record3$

20    IF VAL(RIGHT$(record1$,4)) < VAL(RIGHT$(record2$,4))
      THEN  temp$ = record1$:
            record1$ = record2$:
            record2$ = temp$

30    IF VAL(RIGHT$(record2$,4)) < VAL(RIGHT$(record3$,4))
      THEN  temp$ = record2$:
            record2$ = record3$:
            record3$ = temp$

40    IF VAL(RIGHT$(record1$,4)) < VAL(RIGHT$(record2$,4))
      THEN  temp$ = record1$:
            record1$ = record2$:
            record2$ = temp$
```

String functions : input (INKEY, INKEY$, GET, GET$)
These are functions that have been used liberally throughout
the text so the treatment here need not be over long. The
functions INKEY and INKEY$ wait a specified time (in
hundredths of a second) to see if a key has been pressed. If
a key has been pressed before the end of the time interval
given, then INKEY returns the ASCII code of the key pressed
and INKEY$ the character code of the key pressed. If a key
has not been pressed by the end of the given time, INKEY
returns -1 and INKEY$ an empty string. Thus

```
10    keypress = INKEY(200)
20    keypress$ = INKEY$(200)
30    PRINT  keypress, keypress$
```

This program prints one of

the code for A	if the A key is pressed once within two seconds
-1 followed by A	if the A key is pressed once, later than 2 seconds after RUN, but within 4 seconds
the code for A followed by A	if the A key is pressed once within 2 seconds and once again within the remainder of 4 seconds
-1	if no key pressed within 4 seconds

The following is another illustration of INKEY and could be
the numeric control fragment of one 'reel' of a fruit
machine program.

```
 5    CLS
10    number = 0
20    REPEAT
30      PROCdelay
40      number = (number + 1) MOD 10
50      PRINT TAB(20, 12); number
60      key$ = INKEY$(10)
70    UNTIL key$ = " "
80    END
```

The goal here might be to press the space bar and freeze a
large number on the screen.
INKEY can also be used to test whether a particular key

is pressed at the instant the function is called. This facility is controlled by using a negative parameter in INKEY. For details of how to do this, consult the User Guide.

The difference between the GET pair of functions, and the INKEY pair, is that the GET pair wait until a key is pressed, whereas the INKEY functions always give up after the specified time has elapsed, and continue executing the program. A common application of GET is in file listing. The computer prints a page from a file, then prompts the user with the question "more?". A "yes" answer causes printing to continue, a "no" answer stops the printing.

```
10      REPEAT
20        PROCprintapage
30        PRINT "more?"
40        keypress$ = GET$
50      UNTIL INSTR("Nn", keypress$)
```

Exercises

1 Write a program that counts the number of words containing four letters, in a sentence. You will find that the structure is simplified if words are separated by exactly one space and a sentence is terminated by a full stop preceded by a space.

2 Write a program that checks an entered line of text and prints a message indicating whether or not it satisfies the spelling rule: "i" before "e" except after "c".

3 Write a program that reports the longest word in a sentence.

Chapter 9 *Graphics*

In this chapter the exciting subject of computer graphics is explained in more detail. Although a sound basis of the foundation techniques in this fascinating subject is given, a complete coverage of computer graphics is beyond the scope of this text. The BBC computer incorporates many of the features found in more exotic and expensive processors and all the facilities available on the machine are explained at an introductory level.

The limitations of the BBC computer for advanced graphics are, of course, lack of processor power and secondly, a reduction in colour availability with increasing plotting accuracy. However the available facilities will support a wide variety of graphics applications. As well as dealing with straightforward plotting techniques we look at plottng involving logical operations (which gives the ability to divide images into separate planes), windowing, and basic interaction techniques from the keyboard. Some mathematics concerning shape generation is used in the chapter. You can ignore this with impunity if you are unhappy with it.

A logical presentation of the various PLOT, VDU and GCOL facilities means that these cannot be dealt with consecutively as they are presented in the User Guide. Rather they are introduced as required. Indeed some uses of the VDU statement have been introduced in a previous chapter and more are dispersed throughout this chapter.

9.1 Raster Scan displays

Although it is not strictly necessary to acquire an understanding of the functioning of the display hardware in your computer to write graphics programs, a rudimentary knowledge will make it easier to understand why display instructions are organised as they are. It will also make you appreciate the limitations of your machine and this will be especially useful when you come to do animation.

In a raster scan display a picture or image on the screen is organised as a collection of individual elements called 'pixels' or 'pels' in image-processing jargon. The number of pixels that the screen is divided into determines the accuracy or resolution to which we can draw pictures. Information about the colour of each pixel is stored in a refresh buffer memory. The picture on the screen is created

by hardware that repeatedly scans this memory and colours the corresponding points on the screen. This scan takes place 50 times a second and any change made to the stored image causes an instantaneous change on the screen.

```
00000000000000000000000000000
00000000000100000000000000000
00000000000101000000000000000
00000000010001000000000000000
00000000010000010000000000000
00000001000000010000000000000
00000010000000000100000000000
00000100000000000010000000000
00001111111111111100000000000
00000000000000000000000000000
```

information stored in a raster scan memory for
a black/white or on/off triangle image

A raster scan display should not be confused with a random scan display, (a much more expensive device) in which a picture of, say, a triangle would be stored as three coordinate pairs. A raster scan beam can only move horizontally from left to right. It needs as many memory elements as there are pixels on the screen and there is a one-to-one correspondence between the so-called refresh buffer memory and the pixel image. A line in a picture has to be drawn on the screen as a series of bright pixels that merge into one another. Whether or not this gives a satisfactory impression depends on the resolution. If, for example, we were operating in MODE 4 (320 pixels horizontally by 256 pixels vertically) then the memory can be imagined as a two-dimensional array of 320x256 elements.

```
            320
      0000000000000000000          ....................
      00000000010000000000         ..........+.........
      00000000101000000000         .........+.+........
      00000001000100000000         ........+...+.......
256   00000010000010000000         .......+.....+......
      00000100000001000000         ......+.......+.....
      00001000000000100000         .....+.........+....
      00011111111111110000         ...++++++++++++.....
      00000000000000000000         ....................
```

refresh buffer   display + = bright
memory   . = dark

Each memory element in this mode can only store a 1 or 0 and

so the display is an on/off or two colour display. A row of memory elements in the refresh buffer memory corresponds to a horizontal band on the screen and the hardware continually scans through the memory, as the beam moves across the screen and provides an intensity modulated or on/off beam signal to drive the TV or monitor display. The refresh buffer memory is continually being scanned row by row, each row corresponding to a horizontal band on the display screen. The stream of bits that comes out of the memory has to be converted into a continuous or analogue signal that controls the beam intensity. The normal TV scanning and deflection electronics are used and synchronise or control the outflow of bits from the memory.

The memory requirement for MODE 4 is thus 320x256 bits (80K bits or 10K bytes). For an explanation of the terms bit and byte see Appendix 5. If we operate in MODE 5, the spatial resolution drops to 160x256. We can therefore imagine the memory to be organised as a two-dimensional array of 160x256 elements. This time, however, each pixel can be one out of 4 colours and this information needs 2 bits/pixel. (2 bits can be used to specify 4 states 00, 01, 10 and 11). Thus, although we have contracted the spatial resolution, the memory requirement is still 10K bytes (2x160x256 bits). MODE 5 memory organisation can be imagined as two planes of 1-bit memory cells.
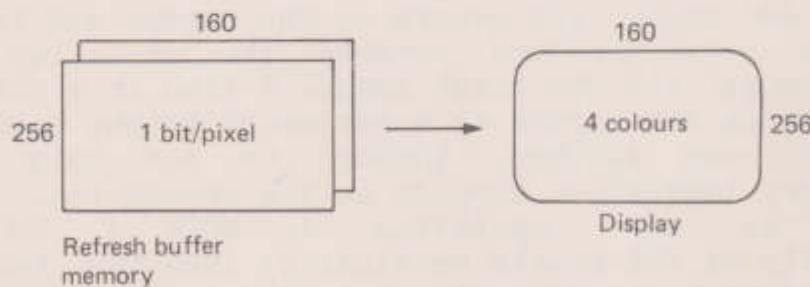
or as a single plane of 2-bit memory cells.
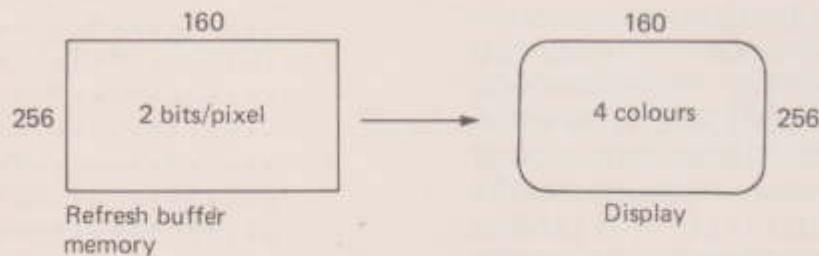
Both concepts of the memory organisation - n planes of 1-bit cells, or 1 plane of n-bit cells - are important in graphics programming.

9.2 Screen coordinates

As already mentioned, the screen is organised into a two-dimensional array of pixels at a particular spatial resolution that depends on which mode is selected. We saw in Chapter 1 that, in a BASIC program, we specify points on the screen by using (x,y) coordinates. The question now is: what values of (x,y) correspond to a particular pixel? The answer to this may seem rather unusual but we shall explain the reason for it. The screen coordinate system used in a BASIC program is completely independent of mode and is a notional or imaginary system of 1280x1024, with (0,0) in the bottom L.H. corner. Thus to refer to a point in the top R.H. corner of the screen we would use (1279,1023). To refer to a point at the centre of the screen we would use (640,512). These coordinate addresses can be used in any of the graphics modes: 0,1,2,4 and 5. The display software then internally scales a point in the notional coordinate system into the mode system with a reduction in spatial resolution that depends on the mode selected. Thus the programmer always works in a 1280x1024 system and the display software rescales the program coordinates into the mode coordinates according to whatever mode is being used. The physical size of a pixel on the screen depends on mode number. In MODE 0 (640x256) the following program coordinates will refer to the same pixel - the one in the top R.H. corner

1278,1023	1279,1023
1278,1022	1279,1022
1278,1021	1279,1021
1278,1020	1279,1020

Thus executing the following will cause only one pixel to light up:

```
PLOT 69, 1278, 1023
PLOT 69, 1279, 1023
PLOT 69, 1278, 1022
PLOT 69, 1279, 1022
PLOT 69, 1278, 1021
PLOT 69, 1279, 1021
PLOT 69, 1278, 1020
PLOT 69, 1279, 1020
```

(PLOT is explained below and PLOT 69, x, y means plot a

single point at (x,y).) Conversely executing any one of these statements causes the same pixel to light up. You can see from this that in MODE 0 there are 8 pairs of program coordinates referring to 1 screen pixel. Similarly in MODE 1 (320x256) there are 16 pairs of program coordinates for each screen pixel. To refer to a screen pixel only one pair of program coordinates for that pixel needs to be used.

The reason for doing things this way is that a program can be run at different resolutions by simply changing the MODE statements. A program that runs in MODE 1 will run in MODE 0 at a higher resolution because the same notional coordinate system has been used in both cases. The shape of graphics images will be preserved (subject of course to the limitation that changing resolution does in itself result in a shape change).

The pixel geometry with respect to the MODE resolution is as follows:

MODE 0 : 640 x 256
8 points/pixel
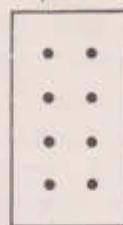
MODE 1 : 320 x 256
16 points/pixel
(also MODE 4)
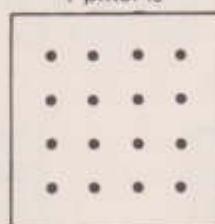
MODE 2 : 160 x 256
32 points/pixel
(also MODE 5)
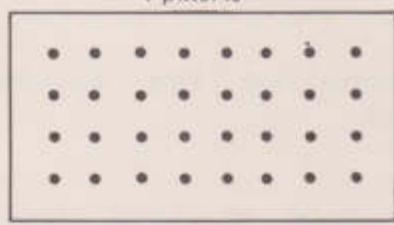
To demonstrate the changing pixel geometry try running the following:

```
10    FOR mode = 0 TO 2
20      MODE mode
30      PLOT 69, 640, 512
40      keypress = GET
50    NEXT mode
```

Pixel separation

Another experiment that you should try demonstrates the ability or inability of your display monitor to separate pixels at the required resolution. The most 'complex' or 'busiest' pattern that we can plot is a checkerboard pattern of alternating bright dark, where each checkerboard element is 1 pixel. To generate one line of such a pattern in MODE 4 we would proceed as follows:

```
10    MODE 4
20    scale = 4
30    FOR x = 0 TO 1279 STEP scale*2
40      PLOT 69, x, 512
50    NEXT x
```

PLOT 69 plots a bright point. We step 'scale*2' each time because we are brightening up every second pixel. Now if you are using a domestic TV you will find that it is difficult if not impossible to see the blank pixels between the bright or on pixels. You will get a greyish continuous line with a hint of the on/off alternation. Adding

```
60    MOVE 0, 500
70    DRAW 1279, 500
```

will demonstrate the difference between this pattern and a continuous line. Now the resolution of this pattern can only be improved by using a monitor. This effect must therefore be borne in mind if you intend using individual pixels as significant dots to make up patterns.

The next fragment does the same thing in MODE 5 in alternating colours black, red, yellow, blue, black, etc. This time you can see that there is a colour resolution limitation - the colours tend to spill into each other.

```
10    MODE 5
20    scale = 8
30    VDU 19, 3, 4, 0,0,0
40    colour = 0
50    FOR x = 0 TO 1279 STEP scale
60      GCOL 0, colour
70      PLOT 69, x, 512
80      colour = (colour + 1) MOD 4
90    NEXT x
```

The use of GCOL 0 was explained in Chapter 1. Line 80 resets 'colour' to 0 when its value reaches 4.

9.3 The PLOT statement : introduction
The PLOT statement is a multi-purpose line and point plotting procedure.

        PLOT k, x, y

plots to a new point specified by x and y. The type of plotting that takes place is determined by the value of k. The simplest uses of PLOT involve values of k from 0 to 7:

   k = 0   Move relative to the last point. "Move" means
            don't plot a visible line.

   k = 1   Draw a line relative to the last point in
            current graphics foreground colour.

   k = 2   As 1 but in logical inverse colour (this is
            not necessarily the background colour - see
            Appendix 5).

   k = 3   As 1 but in current background colour.

Options 0 to 3 are relative plotting commands. This means that the x and y quoted in 'PLOT k, x, y' are not absolute coordinates (in the 1280x1024 notional system) but relative coordinates. They specify the distance of the new point from the current point. Values of k from 4 to 7 have the same effect as 0 to 3 but absolute coordinates are quoted.

Higher values of k have other effects. The higher values are made up of a base value that selects a plotting effect plus one of the values 0 to 7 that modify that effect as we have already described. For example the base value 64 means plot a single point. We saw above that k = 5 means plot to a point whose absolute coordinates are given. Thus k = 64 + 5 = 69 means PLOT a single point at a specified absolute position. This is the PLOT 69 facility that has been informally introduced already. Similarly k = 64 + 1 means plot a single point at a position specified relative to the

current position. In other words the series of PLOTs based on the value 64 differ in effect from those based on 0 by only plotting the last point on the line connecting the current and previous points. The most useful base values are

    0   for plotting lines
    16   for plotting dotted lines
    64   for plotting single points
    80   for filling a triangular area with colour (see below)

The value of having a relative plot facility will become obvious as the chapter progresses. If we were drawing a single line from say (640,512) to (940,812) we could either use

    PLOT 4, 640, 512
    PLOT 5, 940, 812

which uses absolute coordinates to establish both the start and the finish of the line; or we could use

    PLOT 4, 640, 512
    PLOT 1, 300, 300

which uses absolute coordinates to establish the start of the line and relative coordinates to draw to the end of the line.
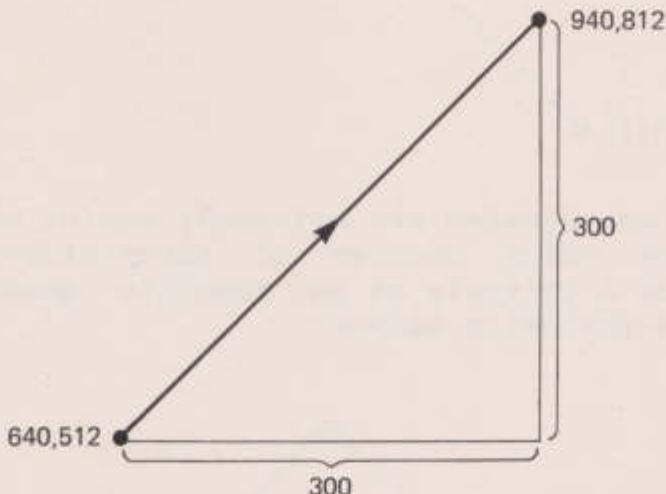
We have already seen that because PLOT 4 and PLOT 5 are likely to be the most commonly used PLOT facilities, these can be written as:

    MOVE x, y   (PLOT 4, x, y)
    DRAW x, y   (PLOT 5, x, y)

Thus all of the following have exactly the same effect:

```
PLOT 4, 640, 512          MOVE 640, 512
PLOT 5, 940, 812          DRAW 940, 812

PLOT 4, 640, 512          MOVE 640, 512
PLOT 1, 300, 300          PLOT 1, 300,300
```

All graphics systems, whether electronic or electromechanical, operate from the 'current position' or CP. The CP is the point arrived at after the execution of a graphics plot statement. For example:

```
MOVE 640, 512      :     CP is now (640,512)
PLOT 1, 300, 300   :     CP is now (940,812)
DRAW 1040, 862     :     CP is now (1040,862)
```
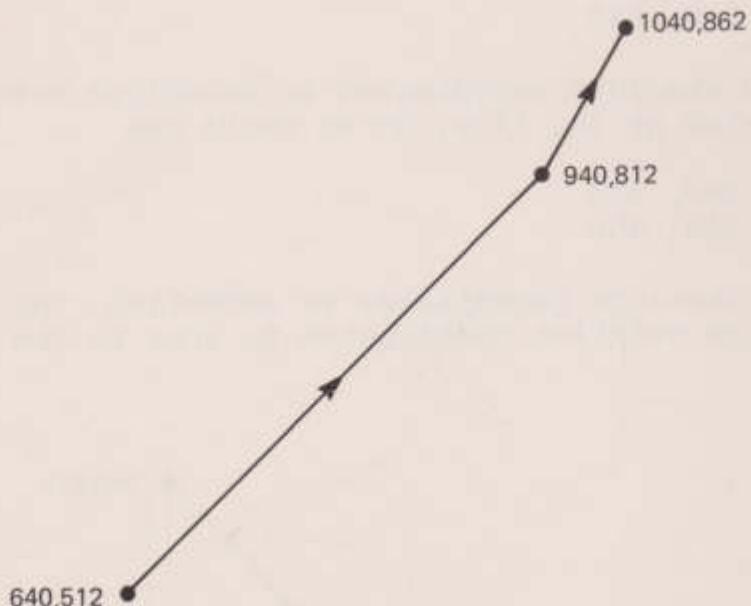
Now relative coordinates are extremely useful when building up a picture using modules or sub-pictures. Suppose we wanted to draw a triangle at any absolute position on the screen in the following manner:

Absolute start position
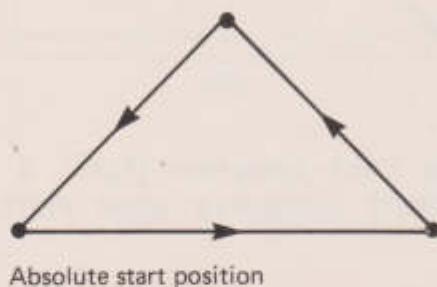
We could define a procedure as follows:

```
100      DEF PROCdrawatriangleat(absx, absy)
120       MOVE absx, absy
130       PLOT 1, 60, 0
140       PLOT 1, -30, 30
150       PLOT 1, -30, -30
160      ENDPROC
```

and this procedure could be called with any absolute coordinates:

```
        PROCdrawatriangleat(640, 512)
```

We could draw 10 triangles on an imaginary diagonal line with the following:

```
 5      MODE 4
10      absx = 0 : absy = 0
20      FOR i = 1 TO 10
30        PROCdrawatriangleat(absx, absy)
40        absx = absx + 30 : absy = absy + 30
50      NEXT i
60      END
```

We could draw triangles that get bigger and bigger by redefining the procedure as:

```
100      DEF PROCdrawatriangleat(absx, absy, size)
110       MOVE absx, absy
120       PLOT 1, 60 * size, 0
130       PLOT 1, -30 * size, 30 * size
140       PLOT 1, -30 * size,-30 * size
150      ENDPROC
```

and calling the procedure with:

```
30      PROCdrawatriangleat(absx, absy, i)
```

where 'i' is the program loop control variable used in the above program. It now controls the size of each triangle. You can perhaps begin to see from these examples the usefulness of having a relative plotting facility. Using procedures to define subpictures is a topic explored in more detail below.

Exercises

1 Write a program to display a selection of the classical optical illusions:
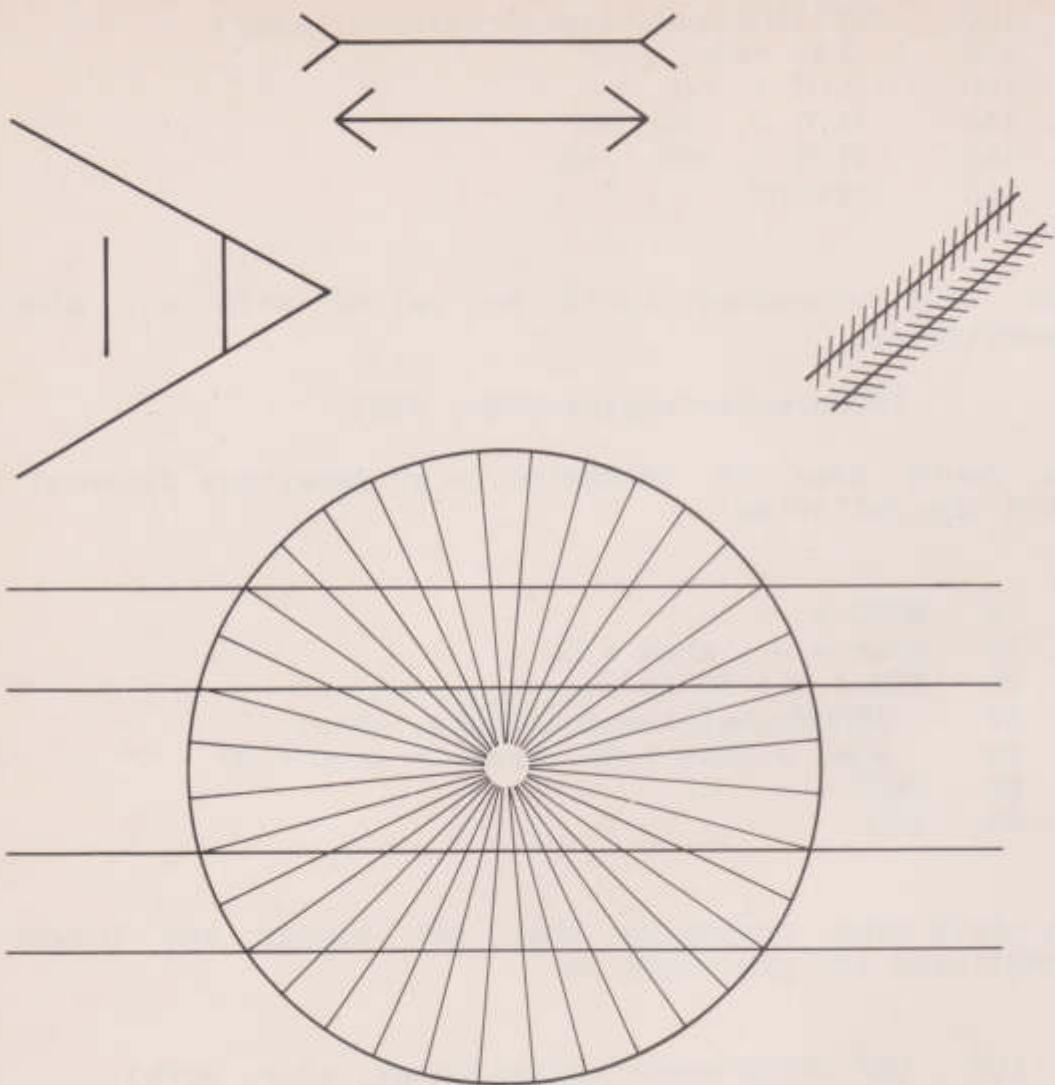
The first two are straightforward, the second two should
be structured using a procedure and loop. Circle
generation was introduced in Chapter 4 (or see below in
this chapter).

2 Write a program to draw the impossible triangle. (Note
that you can structure your program by using the fact
that 6 of the figures are parallelograms and 3 are
quadrilaterals.)
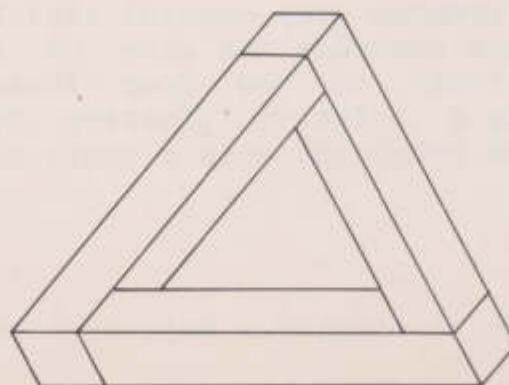
PLOT : simple graphs and origin translation

Drawing graphs is one of the most obvious and common applications of line-drawing statements in computer graphics. We may have a set of 365 daily temperature readings or 12 monthly sales figures and wish to display this data graphically in the familiar way with the variations in the data plotted in the y direction:
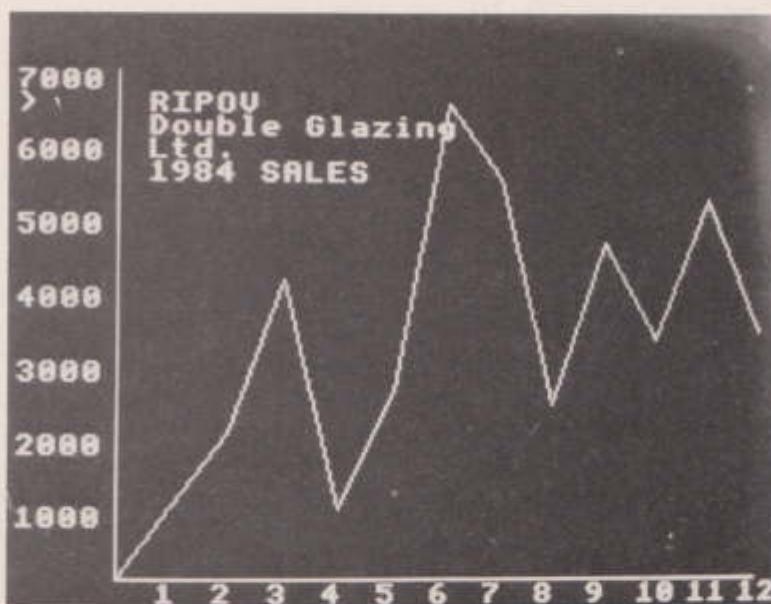
The following produces a first approximation to this:

```
10    MODE 4
20    xscale = 80 : yscale = 1/10
30    DRAW 0, 7000*yscale
40    MOVE 0, 0
50    DRAW 12*xscale, 0
60    MOVE 0, 0

70    FOR month = 1 TO 12
80      READ monthlysales
90      DRAW month * xscale, monthlysales * yscale
100   NEXT month

1000  DATA 1023, 2056, 4132, ...
```

The origin is at the default position of (0,0) and lines 20 to 50 draw the vertical and horizontal axes. Note that after drawing the vertical axis we return to the origin (MOVE 0,0) to draw the horizontal axis. Each monthly sales figure is read from a DATA statement or perhaps in practice from a file. Lines are drawn from one monthly sales figure to the other giving a standard piecewise linear graph. The graph could be started from the correct height for month 1 with

the added complication of:

```
draw the axes

READ monthlysales
MOVE xscale, monthlysales*yscale

FOR month = 2 TO 12

   rest as before
```

The other point to note is the scaling. The vertical extent of the graph must be decided on and the data scaled accordingly. The horizontal extent depends on the desired size and number of data points. In the above program we have used two extra variables to emphasise this point.

The next improvement to be made to the graph is to shift the origin diagonally in towards the centre to make way for legendry along each axis. We can do this by leaving the program exactly as it is and establishing a new graphics origin (0,0) anywhere on the screen by using VDU 29. Suppose we want the new origin to be (256,256). The statement is

```
15    VDU 29, 256; 256;
```

and this single statement added to the above program will plot the same graph in a position nearer the centre of the screen. Incidentally, note the use of semicolons in the VDU statement. A semicolon succeeding a parameter means that the parameter is to be interpreted as a double byte. Two bytes are needed to specify any coordinate in the notional 1280x1024 system.

Graphics legendry

Having created the necessary space for the legendry by translating the origin we can now proceed to add the legendry by continuing the program as follows:

```
110    VDU 5
120    FOR month = 1 TO 12
130      MOVE month*xscale-24, -10
140      PRINT ;month
150    NEXT month

160    FOR sales = 1000 TO 7000 STEP 1000
170      MOVE -150, sales*yscale
180      PRINT ;sales
190    NEXT sales
```

The VDU 5 statement makes any subsequent text printed appear at the graphics CP. The text PRINTs can thus be controlled in the standard way by a FOR statement. Each PRINT is preceded by a MOVE to establish the text position. Note the use of the semicolon in the PRINT statements; this is to override the normal right-justified formatting and ensure that the integers are printed left-justified from the CP. Finally note the negative coordinates in the MOVE command. Remember that the (0,0) origin is now no longer physically at (0,0) on the screen and the negative displacements prevent the text from coinciding with the axes.

Multiple graphs

Another common use of origin translation is the production of a number of graphs on the same screen using the same program module to generate each:
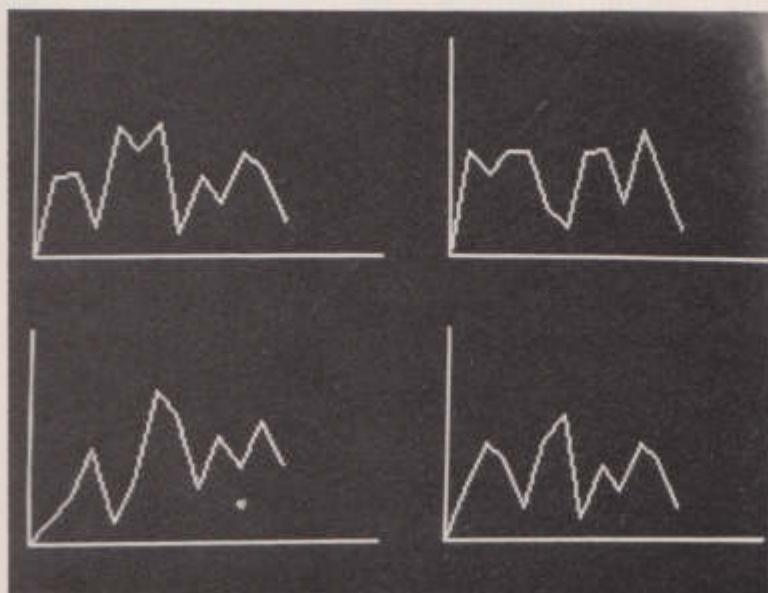
We may want to display, for example, four annual sales characteristics on the same display. In the following program we use exactly the same process as before, now incorporated in a procedure, and call the procedure four times. Each time the procedure is called, its parameters define a new origin.

```
10    MODE 4
20    xscale = 30 : yscale = 1/30
30    PROCdrawgraph(0, 0)
40    PROCdrawgraph(0, 500)
50    PROCdrawgraph(600, 0)
60    PROCdrawgraph(600,500)
70    END
```

```
 80     DEF PROCdrawgraph(ox, oy)
 90       VDU 29, ox; oy;
100       MOVE 0, 0
110       DRAW 0, 400
120       MOVE 0, 0
130       DRAW 500, 0
140       MOVE 0, 0
150       FOR month = 1 TO 12
160         READ monthlysales
170         DRAW month*xscale, monthlysales*yscale
180       NEXT month
190     ENDPROC

200     DATA ... 48 monthly sale figures ...
```

Shape generation and basic mathematical plotting

We shall start this section by illustrating some points concerning the generation of probably the most common mathematically generated shape - the circle. The following generates a 'circle' by plotting a regular polygon of 36 sides. The basic idea was introduced in Chapter 4; here we have simply incorporated the process in a procedure, making the centre and radius parameters.

```
 10     MODE 4
 20     INPUT xc, yc, r
 30     PROCdrawcircle(xc, yc, r)
 40     END

 50     DEF PROCdrawcircle(xc, yc, r)
 60       MOVE xc + r, yc
 70       FOR theta = 10 TO 360 STEP 10
 80         x = r*COS(RAD(theta))
 90         y = r*SIN(RAD(theta))
100         DRAW xc + x, yc + y
110       NEXT theta
120     ENDPROC
```

In this process we are using what mathematicians call polar coordinates. We are working mathematically in polar coordinates - a point is represented by the two values r and theta, and we convert r and theta into Cartesian coordinates x and y using:

$$x = r \cos(theta)$$

$$y = r \sin(theta)$$
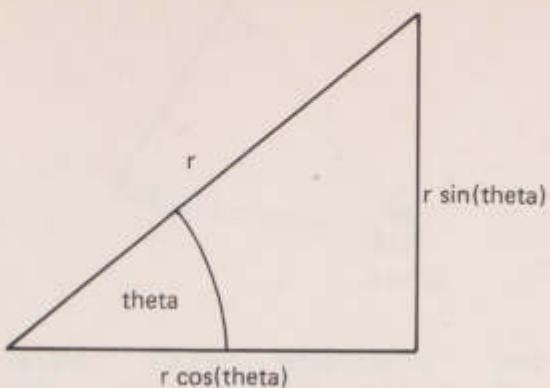
Increments in 'theta' result in new (x,y) coordinates, one (x,y) pair for each vertex on the polygon.
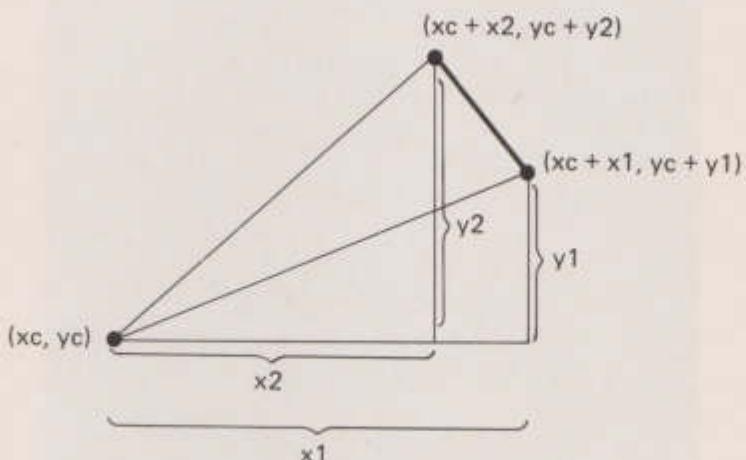
We can make the polygon approach a better and better circle by reducing the FOR loop increments in "theta". Now if we replace line 100 with:

```
100    PLOT 69, xc + x, yc + y
```

then we get a circle comprising 10 dots, one at each vertex, instead of 10 lines. If we add:

```
100    MOVE xc, yc
105    PLOT 69, xc + x, yc + y
```

this has no apparent effect, but instead of moving from vertex to vertex on the polygon we are now moving from the centre along a radius to a vertex each time. We are generating the circle by sweeping the radius rather than than following the circumference. This difference is important in colour fill (later).

180
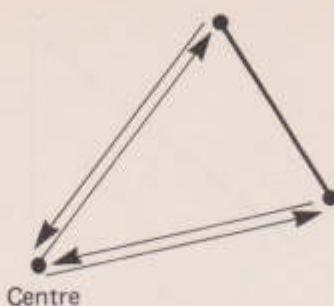
Centre

The next two programs will help develop your understanding of polar coordinates. The first program generates a circle then 6 circles of the same radius centred at 60 degree intervals on the circumference of the first:
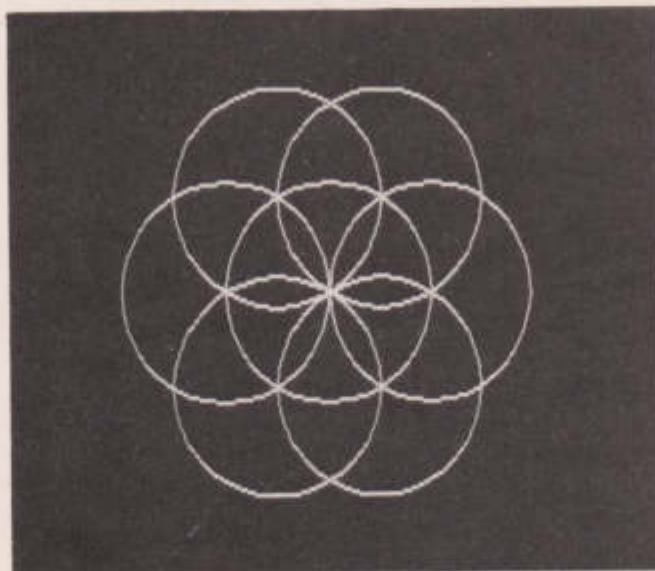

```
10    MODE 4
20    INPUT sx, sy, r
30    PROCdrawcircle(sx, sy, r)
40    FOR circle = 0 TO 320 STEP 60
50      xc = sx + r*COS(RAD(circle))
60      yc = sy + r*SIN(RAD(circle))
70      PROCdrawcircle(xc, yc, r)
80    NEXT circle
90    END

100   DEF PROCdrawcircle(xc, yc, r)
110     MOVE xc + r, yc
120     FOR theta = 10 TO 360 STEP 10
130       x = r*COS(RAD(theta))
140       y = r*SIN(RAD(theta))
150       DRAW xc + x, yc + y
160     NEXT theta
170   ENDPROC
```

Now if we take the above program and constrain the radial sweep of each of the 6 subsidiary circles so that they only appear when they are contained by the first circle (which is no longer drawn), we get the familiar 'petal' diagram:
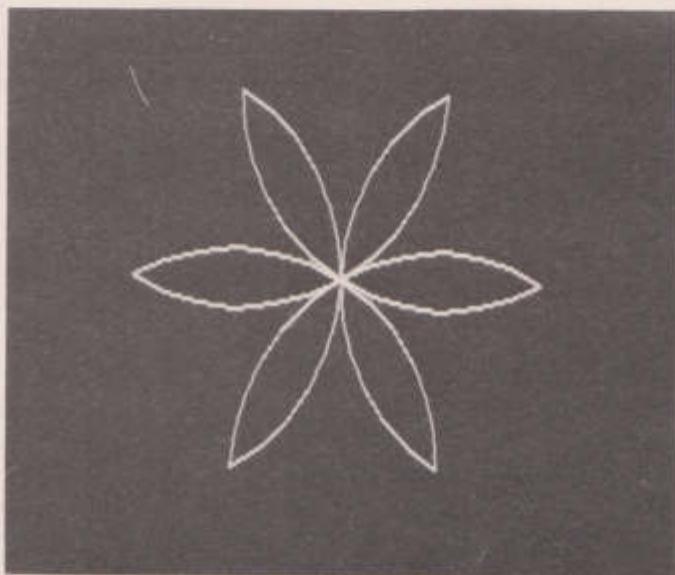

```
10    MODE 4
20    INPUT sx, sy, r
30    t1 = 120 : t2 = 240
40    FOR circle = 0 TO 320 STEP 60
50      xc = sx + r*COS(RAD(circle))
60      yc = sy + r*SIN(RAD(circle))
70      PROCdrawcircle(xc, yc, r)
80      t1 = t1 + 60 : t2 = t2 + 60
90    NEXT circle
100   END

110   DEF PROCdrawcircle(xc, yc, r)
120     x = r*COS(RAD(t1)) : y = r*SIN(RAD(t1))
130     MOVE xc + x, yc + y
140     FOR theta = t1+10 TO t2 STEP 10
150       x = r*COS(RAD(theta))
160       y = r*SIN(RAD(theta))
170       DRAW xc + x, yc + y
180     NEXT theta
190   ENDPROC
```

We leave you the reader to work out the geometry! 't1' and 't2' constrain the circumferences of each of the subsidiary circles, and have to be advanced by 60 degrees each time. We shall be returning to this diagram later to show how to colour fill it.

A more 'complex' shape than a circle is an ellipse. Without worrying about the maths too much the equation for an ellipse is:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$$

where a and b are the semi-major and semi-minor axes. In polar coordinates this gives:

```
r = 1/SQR(cos(theta)*cos(theta)/(a*a) +
                    sin(theta)*sin(theta)/(b*b))
```

The program is just that for the circle with the polar coordinate equation changed and some concession to numerical efficiency made:

```
10    MODE 4
20    INPUT xc, yc, a, b
30    MOVE xc + a, yc
40    FOR theta = 10 TO 360 STEP 10
50      cosine = COS(RAD(theta))
60      sine   = SIN(RAD(theta))
70      r = 1/SQR(cosine*cosine/(a*a)+sine*sine/(b*b))
80      x = r * cosine : y = r * sine
90      DRAW xc + x, yc + y
100   NEXT theta
```

Another common application of mathematical computer graphics is plotting a function of a single variable defined by an equation. This procedure is identical both conceptually and practically to drawing simple graphs of sales data or temperature readings. The only difference this time is that the y data are values given by a mathematical equation. The next program plots that old chestnut – the sine wave:

```
10    MODE 4
20    VDU 29, 100; 500;
30    MOVE 0,0
40    DRAW 1000, 0
50    MOVE 0, -300
60    DRAW 0, 300
70    yscale = 300
80    MOVE 0,0
90    FOR x = 0 TO 1000
100     y = SIN(RAD(x))
110     DRAW x, y*yscale
120   NEXT x
```

Exercises

1 Write a procedure to draw a square. Call this procedure from within a loop that decreases the size of the square and increases its rotation or orientation. You will find that this produces a pattern that appears to consist of interacting spirals, providing that you plot a sufficiently large number of squares.
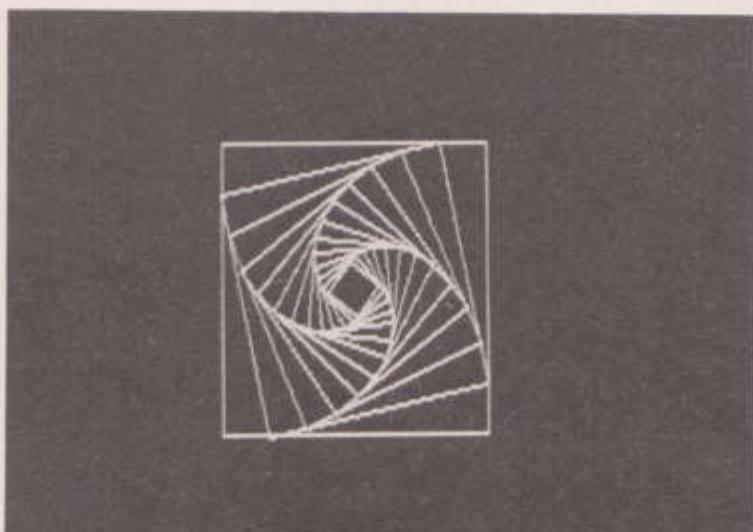
2 Using a nested FOR loop that sets up a grid of origins, fill the screen with a set of such patterns of rotating squares.
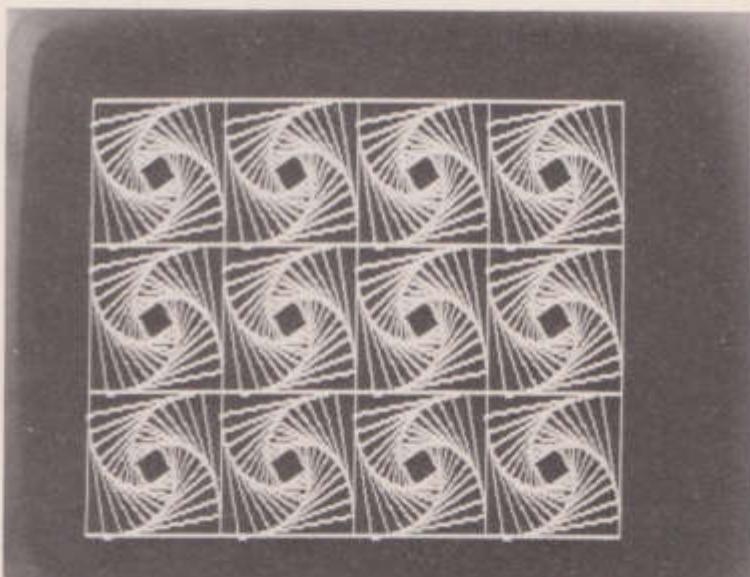
3 Lissajous figures are produced by interacting sine waves controlling coordinate values. For example:

```
x = r * COS(RAD(theta))
y = r * SIN(RAD(theta))
```

produces a circle as we've already seen. Experiment with:

```
x = r * COS(RAD(theta * order))
y = r * SIN(RAD(theta))
```

the interaction of a sine wave with another at 2 times, 3 times, 4 times the frequency.

4 The Lissajous figure plotted using

```
x = a * COS(RAD(theta))
y = b * SIN(RAD(theta))
```

gives a more efficient way of drawing an ellipse than the method using polar coordinates. Write a program to plot an ellipse in this way.

5 Write a program to plot a simple diagram or stylisation of a suspension bridge:
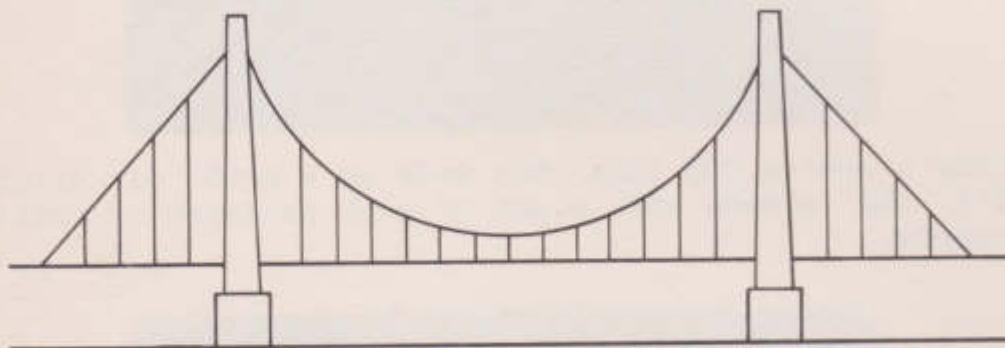
The span between piers is to be an input variable and the pier height proportional to span. The chain curve is a loaded catenary that can be approximated by a parabola:

$$x^2 = 4cy$$
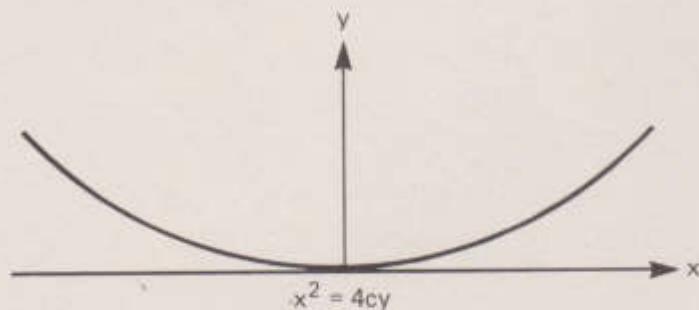
PLOT : windows
As the name implies a window is a definable rectangular area on the screen within which the effect of any (visible) graphics statement is made visible. Graphic statements that have effect outside the window are invisible. The window, if

unspecified, is the screen area (1280x1024). The point of having a window equal to the screen area may not be at all obvious to you if you are new to computer graphics; but it is a vital facility. It means that you can specify coordinates outside the screen area with impunity, and have that part of the resulting image that lies on the screen preserved without distortion. For example specifying a triangle with 2 vertices off the screen would result in:
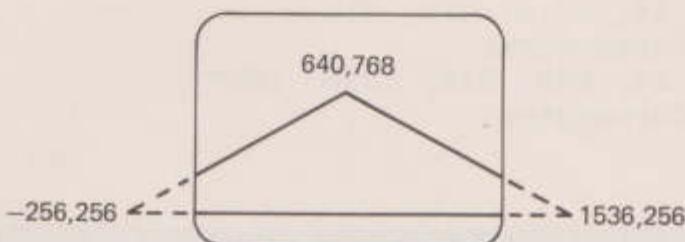
If there were no windowing facility the shape of the visible portion of the figure would be unpredictable and would depend on the magnitude of the off screen coordinates. The use of this facility together with using a window smaller than the screen area is really a topic in advanced computer graphics. It is used in selecting sub-areas of a graphics display for 'magnification' for example and can be used in generating mathematical figures that would be difficult to generate without this facility. We finish off this section with a demonstration of the latter. The next program fills the screen with a spiral. Note that the default screen window comes into effect, the line moving off the screen and back on again.

```
10      MODE 4
20      PROCdrawspiral
30      END

40      DEF PROCdrawspiral
50        r = 10
60        MOVE 640, 512
70        FOR theta = 0 TO 7200 STEP 10
80          r = r + 1
90          x = r* COS(RAD(theta))
100         y = r* SIN(RAD(theta))
110         DRAW 640 + x, 512 + y
120       NEXT theta
130     ENDPROC
```

We can specify the size and the position of a window by using the VDU 24 statement. For example:

    VDU 24, x1; y1; x2; y2;

specifies a window between x = x1 and x = x2; and y = y1 and y = y2. We give the x-y coordinates of the bottom left-hand corner of the window followed by the x-y coordinates of the top right-hand corner of the window. Note the semicolons which are essential. If we now insert two window commands and draw the spiral after each we generate the same pattern as before with two quadrants blanked out:

```
15      VDU 24, 0; 0; 640; 512;
20      PROCdrawspiral
25      VDU 24, 640; 512; 1279; 1023;
30      PROCdrawspiral
35      END
```
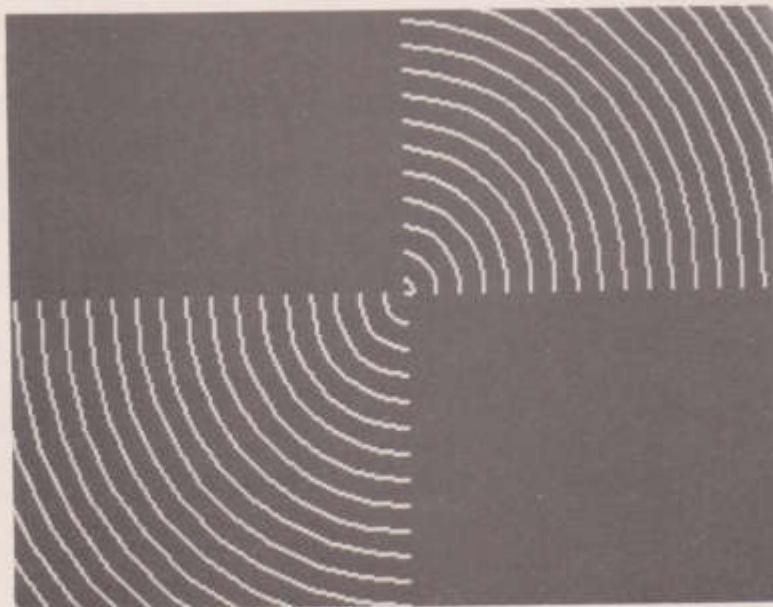

Note that we have to draw the spiral twice - we cannot have two windows active at once. If, for example, we omitted line 20 above, plotting would take place only in the most recently defined window - the top right quadrant. Line 15 would have no effect whatever. Now each time the spiral is plotted processing still takes place 'in the dark regions'. As far as the computer is concerned we are drawing the complete spiral twice.

Colour fill

Filling the interior of a closed figure with a colour can be accomplished by using the triangular fill facility which is yet another PLOT option. Not all closed figures can be filled using this facility and general algorithms exist to fill closed figures irrespective of their shape. However, these will take a much longer time to execute than the triangular fill facility and much can be accomplished using this method as we shall now describe.

The PLOT facilities 0 to 7 are available with triangular fill as 80 to 87 where, for example,

PLOT 85, x, y

would draw a line from the CP to (x,y) absolute and would fill, in the current foreground colour, the triangular area defined by (x,y) and the last two points visited or the last two positions of the CP. For example to draw and fill a rectangle we could use the following:

10   PLOT 1, 100, 0
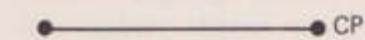
20   PLOT 81, 0, 50

30   PLOT 1, -100, 0

40   PLOT 81, 0, -50
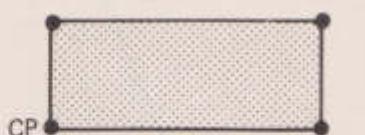
Remember that the CP starts at (0,0). We could accomplish the same effect with:

```
10    PLOT  1, 100, 0
20    PLOT 81, 0, 50
30    PLOT 81, -100, 0
40    PLOT 81, 0, -50
```

but the computer would take slightly longer to do it - part of the area is being filled twice over. We now return to our circle generating module, the one that used a radial sweep. By changing

MOVE xc, yc
PLOT 69, xc + x, yc + y

to

MOVE xc, yc
PLOT 85, xc + x, yc + y

we will generate a filled circle:

```
10    MODE 4
20    INPUT xc, yc, r
30    MOVE xc + r, yc
40    FOR theta = 10 TO 360 STEP 10
50      x = r*COS(RAD(theta))
60      y = r*SIN(RAD(theta))
70      MOVE xc, yc
80      PLOT 85, xc + x, yc + y
90    NEXT theta
```

Similarly our sales graph could be filled by returning to the x axis after each point is plotted. The program is just a generalization of the rectangular plot and fill.
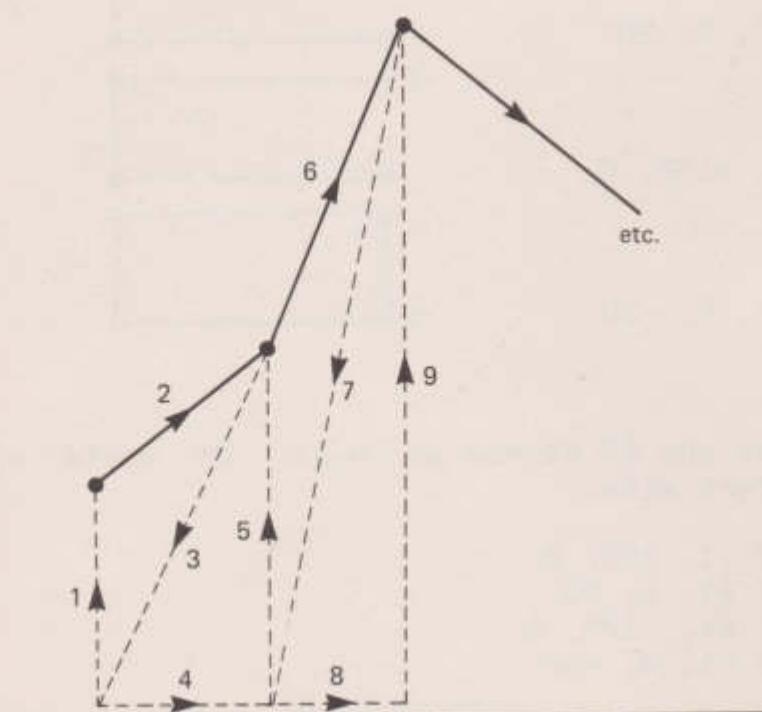

```
10    MODE 4
20    xscale = 80 : yscale = 1/10
30    MOVE xscale, 0
40    READ monthlysales
50    DRAW xscale, monthlysales*yscale : REM STEP 1
60    FOR month = 2 TO 12
70      READ monthlysales
80      x = month*xscale : y = monthlysales*yscale
90      DRAW x,y                          :' REM STEP 2
100     PLOT 81, -xscale, -y              : REM STEP 3
110     PLOT 0, xscale, 0                 : REM STEP 4
120     PLOT 81, 0, y                     : REM STEP 5
130   NEXT month
140   DATA ...
```

The next program generates and fills the 'petal' diagram. It is a useful exercise to follow through the triangular fill sequence as we did with the sales graph fill. The program demonstrates that even quite complicated mathematically generated figures can be filled using the basic triangular fill.
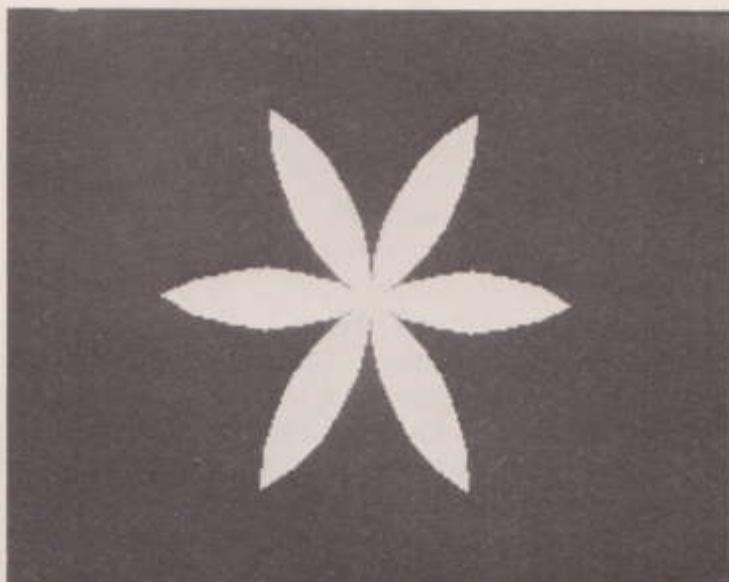

```
10      MODE 4
20      t1 = 120 : t2 = 240
30      INPUT sx, sy, r
40      FOR circle = 0 TO 320 STEP 60
50        xc = sx + r*COS(RAD(circle))
60        yc = sy + r*SIN(RAD(circle))
70        PROCdrawcircle(xc, yc, r)
80        t1 = t1 + 60 : t2 = t2 + 60
90      NEXT circle
100     END

110     DEF PROCdrawcircle(xc, yc, r)
120       x = r*COS(RAD(t1)) : y = r*SIN(RAD(t1))
130       MOVE xc + x, yc + y
140       FOR theta = t1 + 10 TO t2 STEP 10
150         x = r*COS(RAD(theta))
160         y = r*SIN(RAD(theta))
170         x = xc + x : y = yc + y
180         MOVE sx, sy
190         PLOT 85, x, y
200       NEXT theta
210     ENDPROC
```

You can perhaps see from all this that we can generalise and say that any regular or irregular polygon can be filled using triangular fill by the simple expedient of moving to any point within the interior before plotting to each

vertex. Also we must start from a vertex.
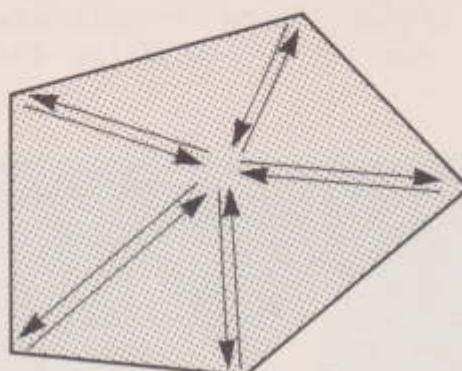
Thus a contour map plotted from a DATA statement could be easily filled. There is a snag! Choosing any interior point will work only if the polygon is convex; this means that if it is a contour map of an island it must not contain any estuaries or else part of the sea may get filled. For some non-convex shapes, a suitable interior point can be found with some care, but for others no single interior point will do. The methods for overcoming this snag are a little tedious. Thus to plot a contour map or convex irregular polygon unfilled:

```
10    MODE 4
20    READ noofvertices
30    READ xs, ys
40    MOVE xs, ys
50    FOR i = 2 TO noofvertices
60      READ x, y
70      DRAW x, y
80    NEXT i
90    DRAW xs, ys
```

To plot a polygon filled:

```
 ....  find an interior point (xi,yi) ....
100    READ xs, ys
110    MOVE xs, ys
120    FOR i = 2 TO noofvertices
130      READ x, y
140      MOVE xi, yi
150      PLOT 85, x, y
160    NEXT i
170    MOVE xi,yi : PLOT 85, xs, ys
```

where (xi,yi) is an interior point. Working out an interior point of course requires an a priori scan of the data. The centre of the minimum enclosing rectangle for example would suffice. To fill in contours within contours in different colours you can adopt the same approach and include GCOL facilities introduced below.

Exercises

1 Repeat the impossible triangle program but this time colour fill the faces.
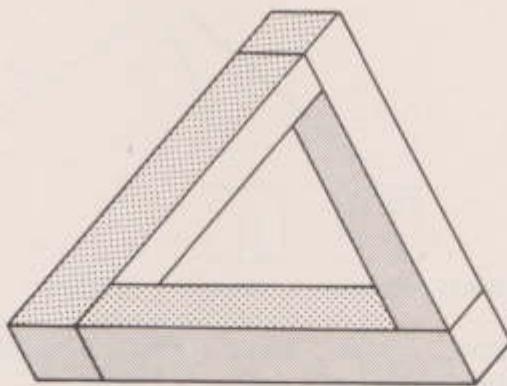
2 Extend the irregular polygon fill program (above) to include a scan of the data to find an interior point.

3 Draw a checker board or games board pattern using colour fill.

Object hierarchy

A large set of line drawings exhibit a hierarchy that can be usefully and profitably reflected in programs that produce line drawings. An electronic diagram, for example, is made up of resistors, capacitors and transistors. A resistor is made up of a line plus a rectangle plus a line:
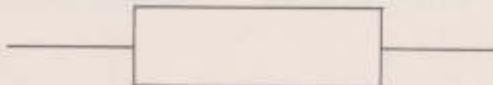
A rectangle is made up of four lines. This may seem a rather trivial statement of the obvious but such an approach can lead to powerful programming techniques. Let's look at another simple example that may demonstrate the point. A child's stylisation of a house (which is after all a simple reflection of an adult's stylisation), can be structured hierarchically as follows:

house

house is : roof   on top of   front

front is: windows   door   inside   outline
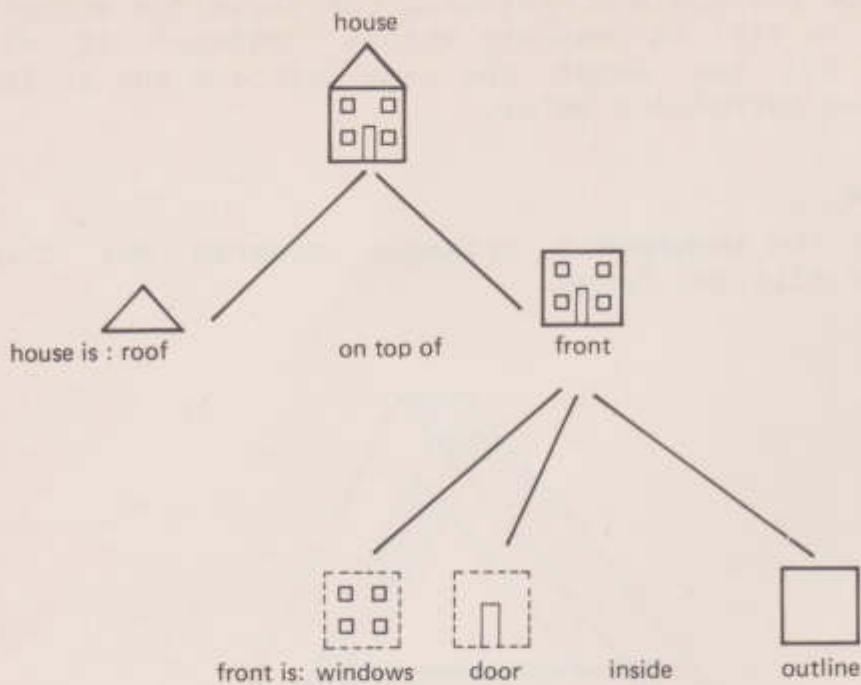
You can perhaps see from this that windows, doors and
outlines are all rectangles of different sizes bearing
certain spatial relationships to each other and can all use
the same program module or procedure. The object hierarchy
in the above tree is reflected in the procedure hierarchy in
the next program.

drawahouse

drawaroof   drawafront

MOVE   drawawindow   drawadoor   drawabox
PLOT

drawabox   drawabox   MOVE
drawabox   PLOT
drawabox
drawabox

MOVE   MOVE
PLOT   PLOT
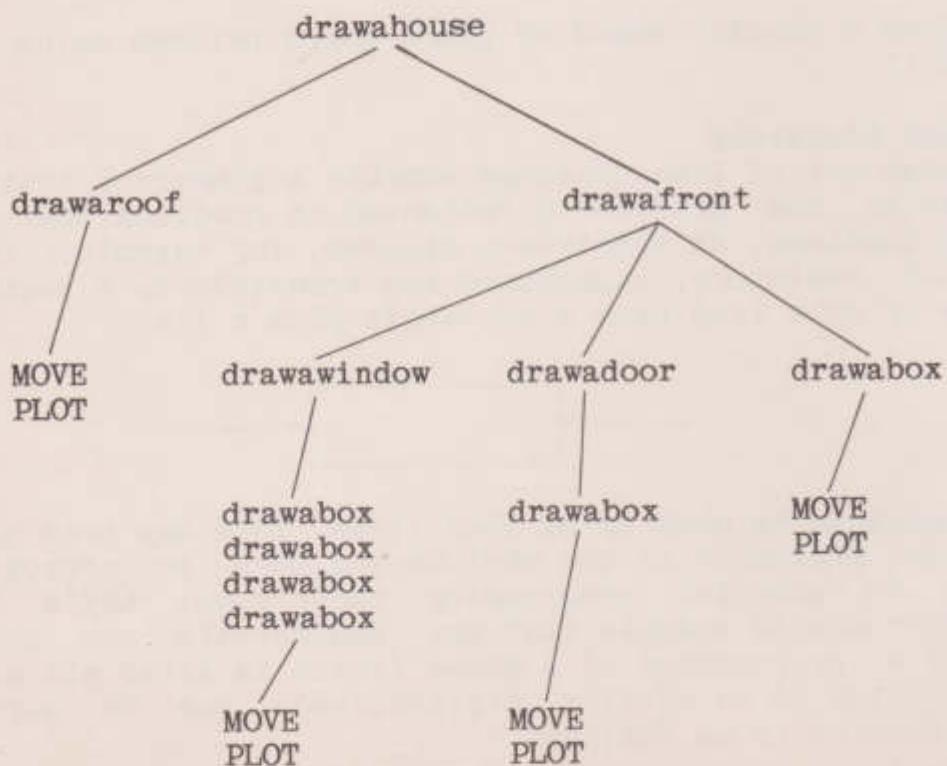
```
10      MODE 0
20      x = 0 : y = 0
30      FOR house = 1 TO 6
40        MOVE x, y
50        INPUT hs, vs
60        PROCdrawahouse(hs, vs)
70        x = x + hs * 30
80      NEXT house
90      END

100     DEF PROCdrawahouse(hs, vs)
110       PROCdrawafront(hs, vs)
120       PROCdrawaroof (hs, vs)
130     ENDPROC

140     DEF PROCdrawafront(hs, vs)
160        PROCdrawabox(hs*30, vs*30)
170        PROCdrawwindows(hs, vs)
180        PROCdrawadoor
190     ENDPROC

200     DEF PROCdrawabox(h, v)
210       PLOT 1, h,0
220       PLOT 1, 0, v
230       PLOT 1, -h, 0
240       PLOT 1, 0, -v
250     ENDPROC

260     DEF PROCdrawaroof(hs, vs)
270        PLOT 0, hs*30, vs*30
280        PLOT 1, -hs*15, vs*15
290        PLOT 1, -hs*15, -vs*15
300     ENDPROC

310     DEF PROCdrawwindows(hs, vs)
320       wh = hs*30/7 : wv = vs*30/5
330       PLOT 0, wh, wv
340       PROCdrawabox(wh, wv)
350       MOVE x, 0
360       PLOT 0, wh, 3*wv
370       PROCdrawabox(wh, wv)
380       MOVE x, 0
390     ENDPROC

400     DEF PROCdrawadoor
410     ENDPROC
```

Thus the procedure 'drawahouse' contains two procedure calls - 'drawaroof' and 'drawafront'. 'Drawaroof' does not need any further definition and contains only references to PLOT. 'Drawafront' contains three procedure calls - 'drawwindows', 'drawadoor' and 'drawabox'. 'Drawwindows' itself contains 4

calls of 'drawabox'. 'Drawabox' contains only MOVE and PLOT. Technically MOVE and PLOT are primitives or procedures that cannot be further defined and are the leaves of the tree. The program builds a hierarchy on MOVE and PLOT that reflects the hierarchy in the picture. We can only scratch at the surface of this particular topic and the power of the approach. This particular program will generate a street of six houses; the user types six pairs of data: the horizontal and vertical scale for each house. (To keep the program length down we have missed out two windows and the door procedure is empty.)
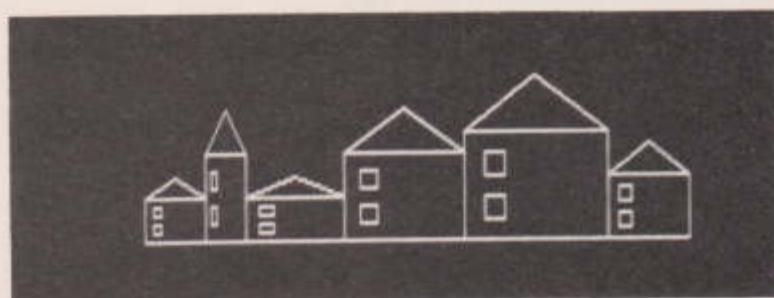
Finally although advanced techniques exist to handle the spatial relationships (windows are a set of four boxes inside another box for example), these are outside the scope of this text and have been handled in an empirical or ad hoc manner, by controlling the start coordinate of each house.

Exercises

1 Complete the house drawing program by adding detail of your choice to the hierarchy. For example windows can contain panes (a loop of boxes within a box), roofs can have chimneys, doors can contain letter boxes etc.

2 Add variety to the above skyline by including different objects that exhibit a similar hierarchy and use the same sub-pictures and primitives. For example an office block is a rectangle containing an array of rectangles; a church is a rectangle and triangle (steeple) with the triangle in a different spatial relationship compared with the house etc.

9.4 The GCOL statement : image planes

The GCOL statement not only sets up the graphic colour as we have already seen, but provides powerful logical processing facilities that can be used in a wide variety of graphics applications. Some of these are developed in the remainder of this chapter and others are utilised in the chapter on animation. Without such logical facilities many advanced graphics applications would be impossible. The GCOL facilities are as follows:

GCOL 0, colour   any subsequent plotting will
                              be in the specified colour

GCOL 1, colour   the colour that results from
                              subsequent plotting is produced
                              by performing an OR operation
                              between the specified colour and
                              the existing screen colour at a
                              pixel

GCOL 2, colour   as 1 but the logical operation is
                              AND

GCOL 3, colour   as 1 but the logical operation is
                              EOR (exclusive OR)

GCOL 4, colour   as 1 but the logical operation is
                              NOT (i.e. the colour at any pixel
                              visited is inverted)

GCOL 0 is straightforward and has already been introduced. The application of logical operations such as OR, AND, EOR and NOT is explained in Appendix 5. The rules concerning foreground and background colour are the same as for the COLOUR statement (colour number greater than 127 defines the graphics background colour).

Incidentally, the statement CLG (CLear Graphics area) can be used to clear the current graphics window and set it all to the current graphics background colour (selected by GCOL). CLS on the other hand clears the current text window and sets it to the current text background colour (selected by COLOUR).

GCOL 1 and 2 have applications in the dividing of an image into planes such as foreground, background and midground and GCOL 3 and 4 have applications in interaction described below.

We shall start by considering multi-plane images. A multi-plane image is an abstraction for the convenience of the programmer. He can build up images independently in planes that are (virtually) separate. This is useful in animation (below) and in being able to deal with a composite image, where different planes have a different priority, e.g. foreground, midground and background (below). We look first at the easier problem of constructing separate planes and switching between them, and then at the more general problem of building up a composite image from planes of different priority.

Virtual image planes : separate images

In a four-colour mode, two bits per pixel are used in the computer refresh buffer memory. However it is up to the programmer how he uses them. We could set up a scheme where we have two 'independent' planes or a scheme where we have a foreground and a midground plane of the same image. Consider the former scheme. We can set up the two bits at a pixel as follows:

```
0 = 00 = image1 and image2 background
1 = 01 = image1 foreground
2 = 10 = image2 foreground
3 = 11 = image1 and image2 foreground
```

The fourth code (3=11) is necessary to signify that for a particular pixel both image1 and image2 planes are 'on'. Starting with the easier consideration of switching between planes (assuming that both images are already built up) we would proceed as follows:

DISPLAY image1:

```
VDU 19, 0, backgroundcolimage1, 0,0,0
VDU 19, 1, foregroundcolimage1, 0,0,0
VDU 19, 2, backgroundcolimage1, 0,0,0
VDU 19, 3, foregroundcolimage1, 0,0,0
```

Image2 is thus set to the background colour selected for image1 and becomes invisible.

DISPLAY image2:

```
VDU 19, 0, backgroundcolimage2, 0,0,0
VDU 19, 1, backgroundcolimage2, 0,0,0
VDU 19, 2, foregroundcolimage2, 0,0,0
VDU 19, 3, foregroundcolimage2, 0,0,0
```

Now image1 is set to the background colour selected for image2 and becomes invisible.

To plot in the image1 plane, say, we have to proceed as follows for each pixel:

```
0 = 00 becomes 01
1 = 01 remains 01 (point already there)
2 = 10 becomes 11 (image2 point already there)
3 = 11 remains 11 (image2 and image1 point
                              already there)
```

The third column is produced by ORing (inclusive) 01 with the second column and we simply precede any plotting statements with the appropriate GCOL statement:

PLOT image1: precede PLOT statements with GCOL 1, 1

Similarly to plot in the image2 plane:

```
0  =  00  becomes  10
1  =  01  becomes  11
2  =  10  remains  10
3  =  11  remains  11
```

and the appropriate GCOL is:

PLOT image2: precede PLOTs with GCOL 1, 2

The following program builds up a simple image in each image plane then repeatedly switches between them.

```
10    MODE 5
20    PROCplotimage1
30    PROCplotimage2
40    FOR screen = 1 TO 10
50      PROCdisplayimage1
60      PROCdelay
70      PROCdisplayimage2
80      PROCdelay
90    NEXT screen
100   END

110   DEFPROCplotimage1
120     GCOL 1, 1
130     PROCdrawacircle(500, 500, 125)
140   ENDPROC

150   DEFPROCplotimage2
160     GCOL 1, 2
170     PROCdrawatriangle(327, 400, 346)
180   ENDPROC

190   DEFPROCdisplayimage1
200     VDU 19, 0, 2, 0,0,0
210     VDU 19, 1, 1, 0,0,0
220     VDU 19, 2, 2, 0,0,0
230     VDU 19, 3, 1, 0,0,0
240   ENDPROC
```

198

```
250    DEFPROCdisplayimage2
260      VDU 19, 0, 4, 0,0,0
270      VDU 19, 1, 4, 0,0,0
280      VDU 19, 2, 0, 0,0,0
290      VDU 19, 3, 0, 0,0,0
300    ENDPROC

310    DEFPROCdrawacircle(xc, yc, r)
320      MOVE xc + r, yc
330      FOR theta = 10 TO 360 STEP 10
340        x = r*COS(RAD(theta))
350        y = r*SIN(RAD(theta))
360        x = xc + x : y = yc + y
370        MOVE xc, yc : PLOT 85, x, y
380      NEXT theta
390    ENDPROC

395    DEFPROCdrawatriangle(xs, ys, s)
400      MOVE xs, ys : DRAW xs+s, ys
410      PLOT 85, xs+s/2, ys+s*0.866
420    ENDPROC

430    DEF PROCdelay
440      TIME = 0
450      REPEAT : UNTIL TIME>100
460    ENDPROC
```

In the above note that we are using completely different
colours in each image. Switching between image planes that
use the same colour is important in animation (Chapter 11).

Incidentally information common to both planes (such as
text, say), need only be plotted once using GCOL 0, 3.

Composite image with priority

Again with a choice of four colours the above scheme can
easily be adapted to set up a three-plane composite image
(foreground, midground and background). Using GCOL the
foreground and midground planes can be independently
accessed and anything drawn in the midground plane that is
shadowed by anything drawn in the foreground plane is
automatically obscured in the composite image. Also we can
delete from the foreground, delete from the midground, add
to the foreground or add to the midground and the
foreground/midground priority is automatically taken into
account. Common operations that we might want to perform
are:

Logical image planes   *Composite display image*

(The figures are meant to be solid or filled)

Initial image—a circle in the foreground against a triangle in the midground
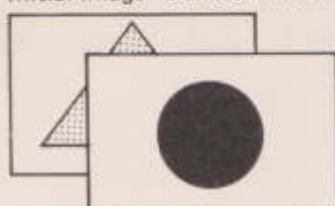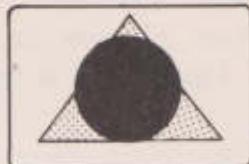
Delete foreground from initial image
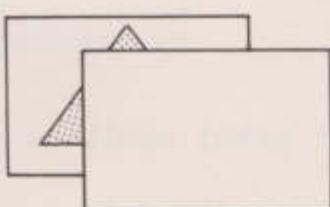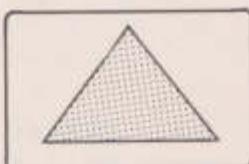
Delete midground from initial image
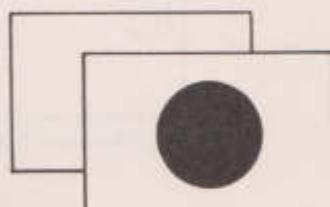
Add to foreground in initial image
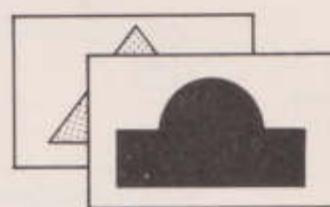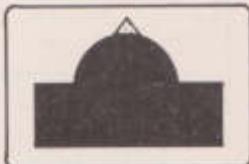
Add to midground in initial image
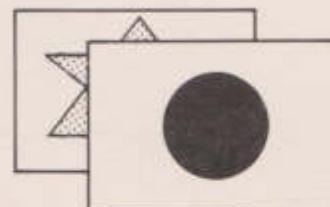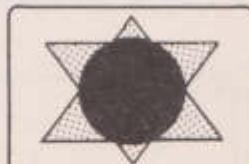
How this is accomplished is now explained. Suppose we are operating in a four colour mode (this allows two planes plus background). A four colour mode means that there are two bits per pixel, i.e. we can imagine the image memory as two one-bit planes.

If the two planes have 0,0 in a pixel position, then the display image is a background point:
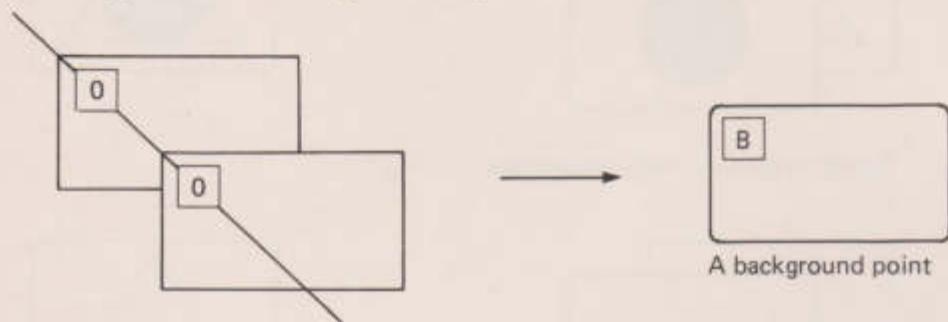
A background point

If the two planes have 0,1 in a pixel position then the display image is a foreground point
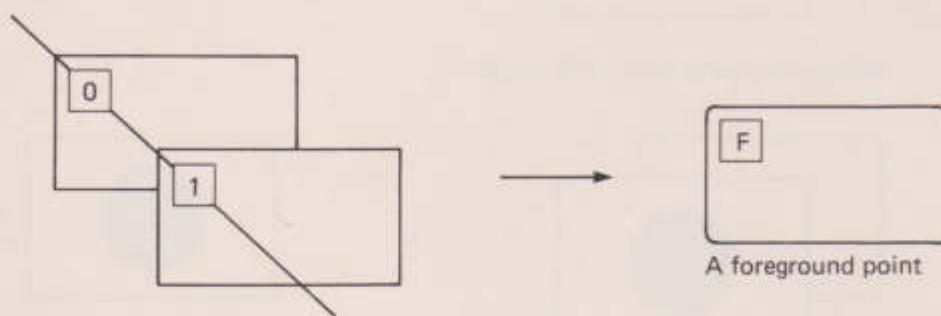
A foreground point

1,0 means a midground point
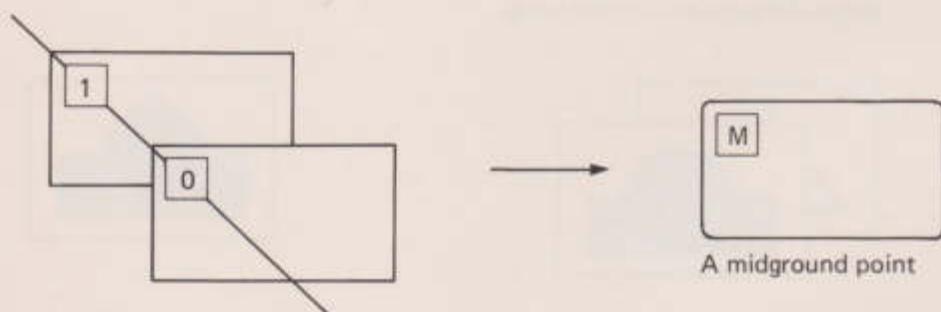
A midground point

Finally 1,1 means a foreground point but this time one that is obscuring a midground point

An (obscuring)
foreground point

Thus we have:

```
0 = 00 = background   point ( . in illustrations)
1 = 01 = foreground   point ( F in illustrations)
2 = 10 = middleground point ( M in illustrations)
3 = 11 = foreground   point ( F in illustrations)
          (obscuring a midground)
```
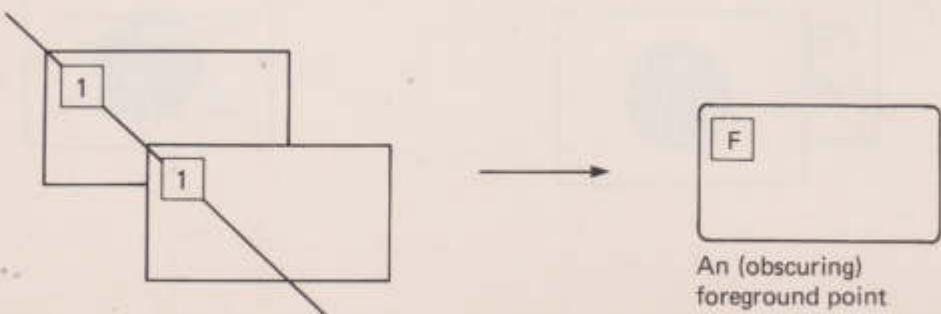
Note that we use two logical colour codes to represent the foreground. This is because we can have 2 types of foreground points - a foreground point obscuring a background point only, and a foreground point obscuring a midground point. We can now give a few examples of 'plane' plotting and you can generalise from these examples.

To PLOT in the foreground

We precede any plot statements with GCOL 1,1 (inclusive OR):

```
GCOL 1,1
PLOT statements to plot figures in foreground plane
```

Now because

```
   00         OR        01        =        01
background          foreground         foreground
```

background points are obscured by foreground points:

```
....................
....FFFFFFFFFFFFF.....
....F................
....F................
....FFFFFFFFFFF.......
....F................
....F................
....F................
....................
```

To PLOT in the midground

We precede any PLOT statements with GCOL 1,2 (inclusive OR):

```
GCOL 1,2
PLOT statements to plot a figure in the midground plane
```

Now because

```
   00         OR        10        =        10
background          midground          midground
```

and

	01	OR	10	=	11
foreground			midground		foreground

background points are obscured by midground points as you
would expect but points that are are already foreground
remain in the foreground colour (but with code 11 indicating
that they are obscuring a midground point). Thus to build up
information in these two planes we use GCOL 1 (inclusive
OR).

```
.....................
....FFFFFFFFFFFFF.....
....FM..........MM.....
....F.M........M.M.....
....FFFFFFFFFFF.M.....
....F...M...M...M.....
....F....M.M....M.....
....F.....M.....M.....
.....................
```

You can perhaps see from this that after a composite set of
planes has been built up any subsequent additions to the
foreground or midground will be incorporated into the
composite image according to their respective priority.

To DELETE from foreground and midground

Now to delete images or parts of images from planes we use
GCOL 2 (AND). To delete a foreground object, we redraw the
object after using GCOL 2,2, where the second parameter
happens to be the midground colour but is used here as a
"foreground delete code". To delete a midground object, we
use GCOL 2,1. For example to delete from the foreground:

```
GCOL 2, 2
PLOT statements to delete figure from foreground plane
```

and the PLOT statements will be exactly the same as the ones
that were used to draw the object being deleted. Now we have

	00	AND	10	=	00
background					background

i.e. background points remain as background

	01	AND	10	=	00
foreground					background

'ordinary' foreground points revert to background

```
10        AND     10      =       10
midground                        midground
```

'ordinary' midground points are left unaltered

```
11        AND     10      =       10
```

'obscured' midground points are now revealed

```
....................
....M............M.....
....MM..........MM.....
....M.M........M.M.....
....M..M.....M..M.....
....M...M...M...M.....
....M....M.M....M.....
....M.....M.....M.....
....................
```

Thus GCOL 2 (AND) can be used to delete and reveal. These operations are now demonstrated. The procedures are left undefined (see earlier sections) but should include colour fill. The following program draws a red circle in the foreground plane and a yellow triangle in the midground plane and any geometrical overlap is automatically taken care of. Note line 20; remember that we use 2 codes for the foreground and these of course should be the same colour.

```
10    MODE 5
20    VDU 19, 3, 1, 0,0,0
25    GCOL 1, 1
30    PROCdrawacircle(500, 500, 125)
40    GCOL 1, 2
50    PROCdrawatriangle(327, 400, 346)
60    END
```

To delete the red circle and reveal any previously hidden parts of the yellow triangle we can add:

```
60    keypress = GET
70    GCOL 2, 2
80    PROCdrawacircle(500, 500, 100)
```

which 'undraws' the circle.

A convenient alternative to the above uses of GCOL 1 and GCOL 2 is often useful. Provided that an object being plotted in a plane does not overlap any object that is already present in that plane, then GCOL 3 can be used for

both the drawing and deleting process. For example to draw an object in image plane 1:

```
GCOL 3, 1
PLOTs etc to draw the object
```

To delete the object we simply repeat exactly the same GCOL and PLOT statements.

On the Model B, in MODE 2, we have 4 bit colour codes (16 colours) and this gives us many more possibilities. One possibility would be to have image planes with:

```
3 foreground colours     (spaceships?)
1 frontmidground colour  (planets?)
1 rearmidground colour   (stars?)
1 background  colour     (sky?)
```

The details of how to do this are beyond the scope of this introductory text but you might try and work out how to do it yourself.

Exercises

1 Undraw a rectangle by working from the centre as if it were a stage curtain being pulled to each wing. Underneath should reveal detail in another colour: legendry that might be used in a caption sequence or anything else you fancy.

2 Plot two pictures using values from DATA statements and then repeatedly read a key. Depending on which key was pressed, display the first picture or display the second picture or display both pictures at once (without re-drawing them).

9.5 Basic interaction techniques (GCOL 3 and GCOL 4)

In this section two interaction techniques are implemented. Both of these use the keyboard, but clearly the principles are the same for either a keyboard or a more convenient device. Both interaction techniques can be used in picture construction and this forms a part of most CAD (Computer Aided Design) systems. Such techniques enable designers to work in a two-dimensional or picture domain. This means for example that an electrical engineer can work with circuit diagrams and an architect with elevations or other projections of buildings, rather than just numbers. Now CAD techniques are an extensive topic by themselves and we shall only be concerned here with picture or line-drawing generation. It is not out of place to examine just briefly how such techniques 'fit in' to CAD programs. A CAD program that accepts a picture as input has to deduce certain information from it. An electrical engineer may draw a

circuit diagram as input. A simple but somewhat unrealistic example serves to illustrate the point; say he inputs a series parallel resistor configuration:
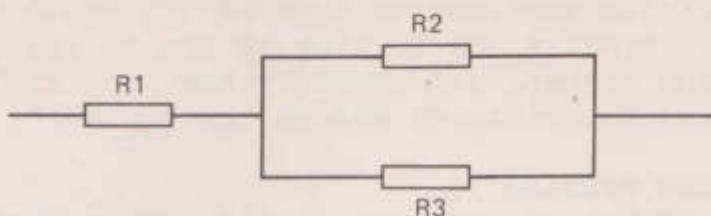
From this the CAD program will have to deduce that a resistor is connected in series to two resistors in parallel and that the total resistance is :

    RT = R1 + R2*R3/(R2 + R3)

It can then evaluate numerical calculations and output required information graphically or otherwise back to the user. The CAD program will also be able to cope with alterations to the diagram - additions, deletions etc.

In the same way an architect may sketch in the elevations of a house and ask for costing, insulation or sunlight calculations. If we return to the circuit diagram, this is something that would be built up using a technique known as 'picking and dragging' (below) A user is presented with a menu of objects and he can pick or select a particular object and drag it to anywhere on the screen:
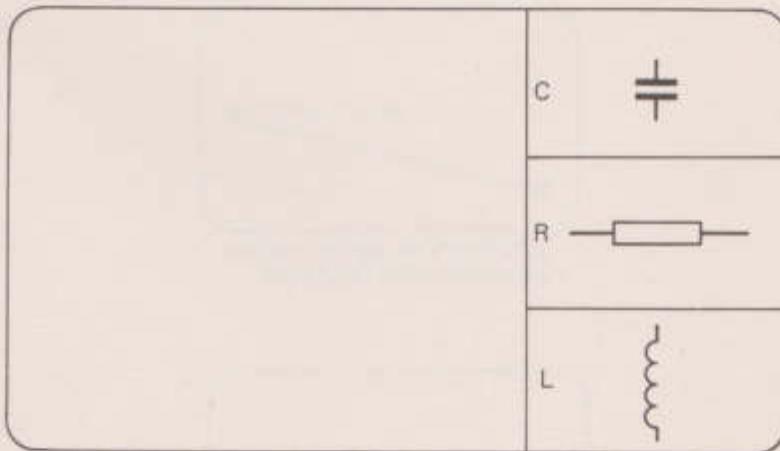
Other operations that might be available on objects are magnification and rotation. Again in the case of an electrical circuit diagram, in parallel with the picture-drawing modules there will be procedures that keep track of the spatial relationship between components. The CAD program can then build up a formula reflecting some required attribute or behaviour of the circuit. This might be

transfer characteristic, frequency response, etc. The computer program's view of the problem is numerical or formula based while the engineer's view remains pictorial. This is a tremendous advantage in most design problems.

In the next two sections we look at the front end of such CAD programs firstly by looking at how we can sketch line drawings on the screen, and secondly how we can pick and drag predefined sub-pictures across the screen.

Rubberband line drawing

Using this technique we can build up a sketch or line drawing on the screen, using line segments whose length and direction are controlled from the keyboard. The program starts off by drawing an arbitrary line from (0,0) to (500,500). By using keys R, L, U, and D (Right, Left, Up and Down) as direction indicators we can move the end point of the line anywhere we want. Key F can be used to 'Fix' the end point of the line.
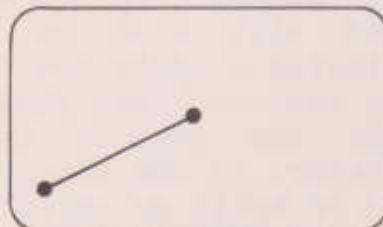
Start of program
Arbitrary line drawn from
(0,0) to (500,500)
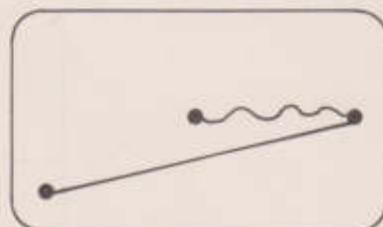
Line endpoint can be moved
anywhere from (500,500)
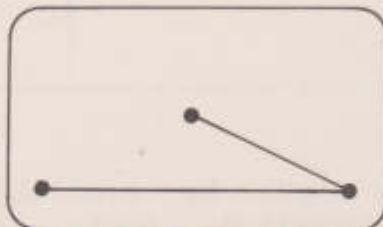
Key F depressed 2nd line arbitrarily
drawn to (500,500) and 1st line
permanently drawn
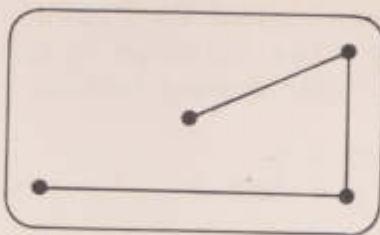
This line can be moved anywhere
and key F depressed again
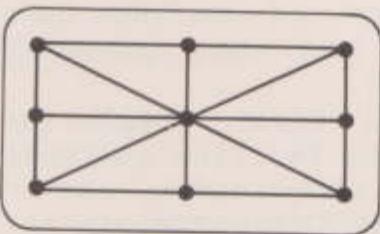
Thus any shape can be built up

The program is as follows:

```
 10     MODE 4
 20     xs = 0 : ys = 0
 30     x = 640 : y = 512
 40     GCOL 3, 1
 50     PROCdrawordelete
 60     REPEAT
 70       command$ = GET$
 80       PROCprocesscommand
 90     UNTIL command$ = "Q"
100     MODE 7 : END

110     DEF PROCprocesscommand
120       IF INSTR("FLRUD",command$) = 0 THEN ENDPROC
130       IF command$ = "F" THEN PROCfix
                            ELSE PROCdrawordelete
140       IF command$ = "L" THEN x = x - 5
150       IF command$ = "R" THEN x = x + 5
160       IF command$ = "U" THEN y = y + 5
170       IF command$ = "D" THEN y = y - 5
180       PROCdrawordelete
190     ENDPROC

200     DEF PROCdrawordelete
210       MOVE xs, ys : DRAW x, y
220     ENDPROC
```

```
230    DEF PROCfix
240      REM Permanent draw to fill in gaps where
250      REM moving line crosses existing lines.
260      GCOL 0,1 : PROCdrawordelete
270      GCOL 3,1
280      xs = x : ys = y
290      x = 640 : y = 512
300    ENDPROC
```

'xs' and 'ys' always represent the start position of the
line currently being drawn and 'x' and 'y' represent the
position of the end of the line being moved. The program
consists of a REPEAT loop that processes commands UNTIL the
key Q (Quit) is typed.

PROCprocesscommand first checks for a valid key. If "F"
has been pressed, then the line currently being operated on
is fixed and the coordinates are set for a new line.
Otherwise a call of PROCdrawordelete is used to delete the
current line in preparation for redrawing it in a new
position. One of the coordinates x, y is updated if one of
the movement keys (L, R, U, D) has been pressed.
PROCprocesscommand terminates by drawing a line to the
position now specified by the x-y coordinates.

The critical statement in the program is GCOL 3, 1
(exclusive OR). This means that lines can be moved over
existing lines without permanently wiping part of them out,
as would be the case without this facility. Normally to
delete an object we would re-plot the object in the
background colour but this would wipe out intersecting parts
of existing lines. Using the above method, an existing line
disappears only momentarily while the current moving line
passes over it.
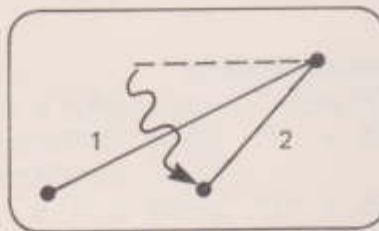
Thus line segment 2 (above) can be swept over existing line
segment 1 without rubbing it out. This can be explained by
reference to the following table.

	1st. DRAW			2nd. DRAW	
old	plotting	new	old	plotting	new
0	1	1	1	1	0
1	1	0	0	1	1

You can see from the bottom row of the table that plotting a 1 on top of a 1 in the first DRAW results in a zero that is restored to a 1 by the 2nd DRAW. The top row of the table gives the effect of a normal draw and erase function. The second DRAW thus erases or undraws, at the same time restoring any holes in existing lines made by the 1st DRAW. We leave it as an exercise to work out why the behaviour is unaltered if GCOL 3 is replaced by GCOL 4.

Picking and dragging an object

We have already mentioned the use of this particular technique above so we'll jump straight in to doing it. In the next program we have set up a triangle in a corner of the screen. This can be moved anywhere using keys L, R, U, and D as before. When the final positioning is achieved key F is pressed to establish the object in that position.

```
10      MODE 4
20      x = 0
30      y = 0
40      GCOL 3, 1
50      PROCdrawatriangle(x,y)
60      REPEAT
70        command$ = GET$
80        PROCprocesscommand
90      UNTIL command$ = "Q"
100     MODE 7 : END

110     DEF PROCprocesscommand
120       IF INSTR("FLRUD",command$) = 0 THEN ENDPROC
130       IF command$ = "F" THEN PROCfix
                            ELSE PROCdrawatriangle(x,y)
140       IF command$ = "L" THEN x = x - 5
150       IF command$ = "R" THEN x = x + 5
160       IF command$ = "U" THEN y = y + 5
170       IF command$ = "D" THEN y = y - 5
180       PROCdrawatriangle(x,y)
190     ENDPROC

200     DEF PROCfix
210       GCOL 0,1 : PROCdrawatriangle(x,y)
220       GCOL 3,1
230       x = 0 : y = 0
240     ENDPROC

250     DEF PROCdrawatriangle(x,y)
260       MOVE x, y
270       PLOT 1, 60, 0 : PLOT 1, -30, 30
280       PLOT 1, -30, -30
290     ENDPROC
```

The program to drag an object is identical to the rubberband program (which drags the end of a line) except that every occurrence of:

    PROCdrawordelete

is replaced by:

    PROCdrawatriangle(x, y)

Picking can be accomplished easily. For example if there were 3 objects, every occurrence of:

    PROCdrawatriangle(x, y)

is now replaced with:

    PROCdrawashape(x, y, pick$)

which is defined as:

```
300    DEF PROCdrawashape(x, y, p$)
310      IF p$="T" THEN PROCdrawatriangle(x, y)
320      IF p$="C" THEN PROCdrawacircle(x, y)
330      IF p$="B" THEN PROCdrawabox(x,y)
340    ENDPROC
```

The variable 'pick$' contains one of the codes "T" for Triangle, "C" for Circle or "B" for Box. At the start of the program 'pick$' is set by

    45   PROCpick

and in PROCfix by

    235   PROCpick

where PROCpick is defined as

```
350    DEF PROCpick
360      PRINT TAB(0,0); "Pick, (T/C/B)";
370      pick$ = GET$
380      PRINT TAB(0,0); "                "; 
390    ENDPROC
```

PROCdrawacircle and PROCdrawabox are straightforward.
    Other common facilities in picking and dragging programs are magnification and rotation. For example in the above dragging program, a 5th key option could be "M" for

'Magnify'.

```
    DEF PROCprocesscommand
             .
             .
             .
    IF command$ = "M" THEN scale = scale*1.2
    PROCdrawatriangle(x, y, scale)
    ENDPROC
```

PROCdrawatriangle would now be defined as:

```
250     DEF PROCdrawatriangle(x, y, scale)
260        MOVE x, y  :  PLOT 1, 60*scale, 0
270        PLOT 1, -30 * scale, 30 * scale
280        PLOT 1, -30 * scale, -30 * scale
290     ENDPROC
```

The variable 'scale' would be initialised to 1 at the start
of the program. Note that the scale is increased by 20 per
cent for magnification so that the increase in the scale is
proportional to the current size of the triangle. Also
'scale' needs to be reset to unity after each triangle has
been moved, magnified and established. We leave the
alterations needed for rotation as an exercise.

Saving a line drawing
An image that has been created by rubberbanding can be saved
as a list of coordinates and subsequently regenerated by a
simple program reading the coordinates from a file and using
DRAW. The coordinates can be saved initially in two parallel
arrays and when the drawing is complete, the array contents
dumped into a file. The coordinate saving should clearly be
part of the 'fixing' process:

```
230     DEF PROCfix
260        GCOL 0,1 : PROCdrawordelete
270        GCOL 3,1
280        line = line + 1
290        xcoord(line) = x
300        ycoord(line) = y
310        xs = x : ys = y
320        x = 640 : y = 512
330     ENDPROC
```

Similarly an image that has been created by picking and
dragging an object can be saved, most economically, using

three parallel arrays. The program would store, for each object, a pair of coordinates followed by a code indicating the class of object drawn at that position. The program would terminate by outputting the contents of the three arrays to a file. The regenerating program would contain the object-generating procedures again called from a shape selection procedure, the appropriate procedure for each shape being selected according to the stored code.

Exercises

1 Improve the rubber band program so that the start coordinate is input from the keyboard.

2 Introduce colour so that the fixed lines are displayed in one colour, but the moving line appears in a contrasting colour.

3 Write a rubber band program where the permanent lines are to be constrained to the horizontal or vertical direction. For example an imperfectly drawn horizontal line:
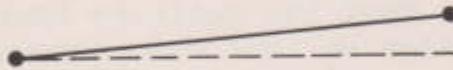
is to be corrected to a perfect horizontal line:
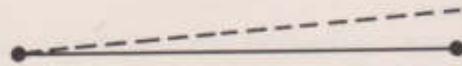
4 Write a picking and dragging program that will allow such diagrams as the following to be constructed:
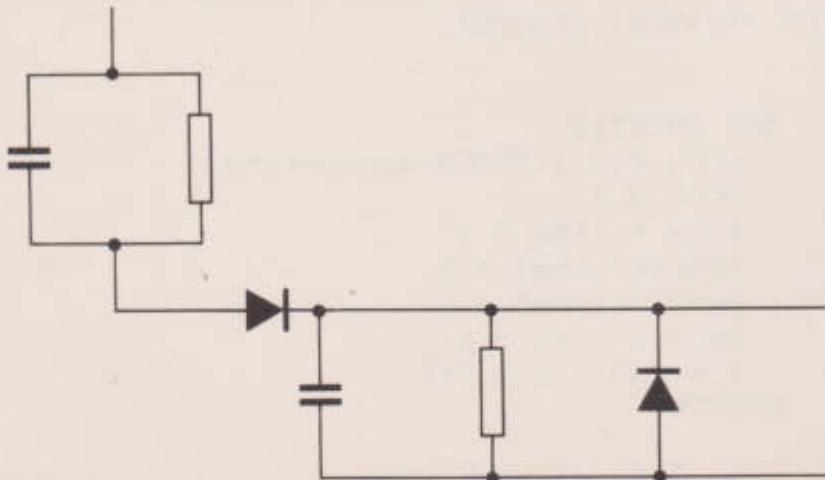
Note that this will have to contain both object dragging
and rotation (0 or 90 degrees only) as well as
rubberband line drawing.

5   Incorporate the picture-filing suggestions in one of your
    programs.

Chapter 10 **Sound**

10.1 Sound and pitch

Perhaps it is best to introduce you to sound by tying up the aural impressions or qualities of various sounds with visual impressions. Any sound is caused by a pressure variation with time, or successive compressions and rarefactions of air. This pressure variation can be converted into a variation of electrical current as a function of time, using a transducer such as a microphone, and the electrical current or signal displayed on an oscilloscope. This gives us an idea of what sounds look like when converted into such a signal. 'Seeing' sounds then provides us with a useful bridge between the aural impressions of sounds and the hard numbers that we have to supply to a computer program in order to to generate the sounds. Various examples are shown in the photographs below.
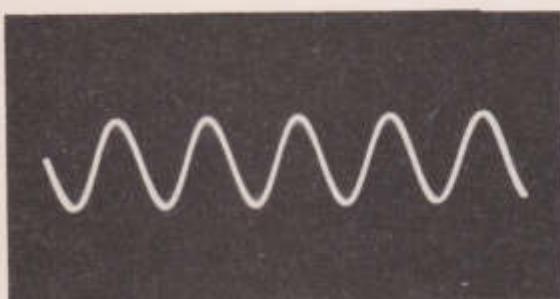
A pure tone: Middle C
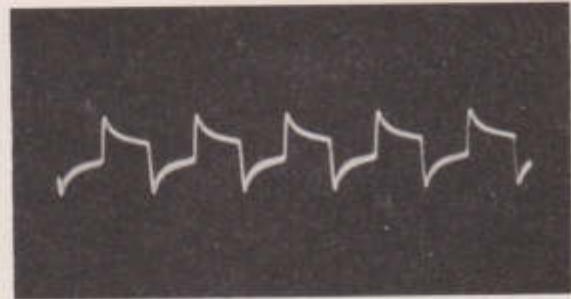
The same tone as generated by the BBC Micro
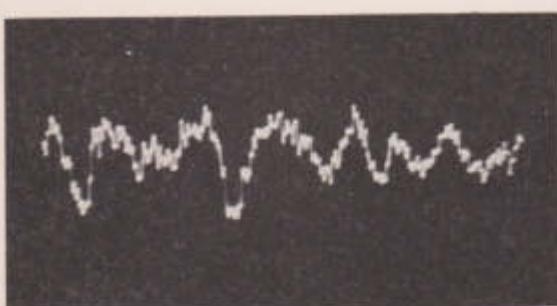
An extract from rock music
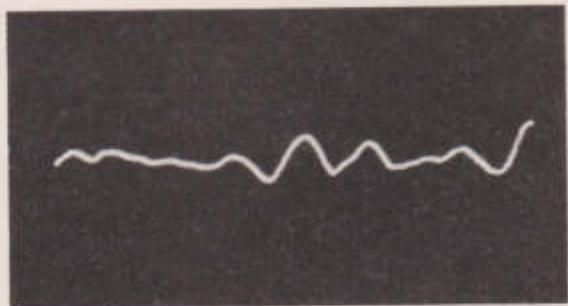
An extract from a nearly monotonic section of Eric Satie's music

A pure tone has a regular sinusoidal appearance and can only be generated perfectly by electronic means or by mechanical

(or electromechanical) devices such as tuning forks. Musical instruments do not generate pure tones but a complex mixture of pure tones, each mixture a function of many variables, the most predominant being the instrument type and the player. An oboe produces a mellow smooth tone, a trumpet a much harsher cutting tone.

The basic sound generated by the BBC micro is not a pure tone either. It is a distorted square wave (see illustration) and this is why it sounds 'electronic'. Its other less than desirable characteristic is that it changes shape as a function of frequency or pitch. This means that the character of the sound changes with frequency.

The BBC micro sound chip can generate tones of specified pitch, and white noise. The ability to combine different sounds and to control transients makes it possible to generate coarse approximations to instrumental sounds. We can also create the cacophonous effects that emanate from arcade games.

To start with, however, we shall concentrate on single tones and examine the facilities available in the SOUND statement. All the sounds made in this chapter can be enhanced by special effects and transient control. This has been left to the end of the chapter because it can only be fully appreciated after the other facilities have been covered. You can however jump ahead, read some of the material on the ENVELOPE facilities, and then use them in earlier programs.

10.2 The SOUND statement

The SOUND statement allows us to manipulate three basic parameters of a single tone sound waveform: the frequency or pitch, the amplitude or loudness and the duration of the sound. A SOUND statement is written as follows:

```
SOUND    C, A, P, D

where    C is the channel number
         A  is the amplitude
         P is the pitch number
         D is the duration
```

If C is set equal to 0 then white noise is produced. This facility is described at the end of the chapter. For tones, C is set to 1, 2 or 3. All four channels can be 'on' simultaneously and the use of channels in combination is explained below.

As we saw in Chapter 1, A, the amplitude, can have a negative value ranging from 0 for silence down to -15 for maximum loudness. The complete range is from -15 to 4 and the use of the 4 positive integers is explained below in the

sections on the ENVELOPE statement. Note that A specifies the amplitude of an electronically generated waveform. Even although A = -14 generates a wave twice the amplitude (or voltage) of a wave for A = -7, it does not sound twice as loud. This is because the human ear has a logarithmic response to loudness. We show below how these parameters relate to the actual waveform generated by the sound synthesizer.
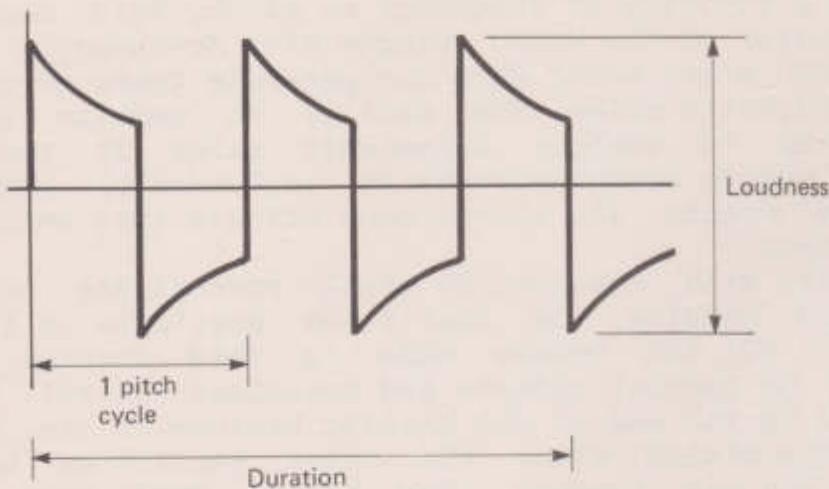
The pitch, or frequency, of a note is the number of undulations or cycles per unit time, where a complete undulation is a positive and a negative hump. The range of basic square wave tones that can be generated on your computer spans 5 octaves and the frequency of each tone is specified by a so-called pitch number, P. (An octave is a doubling of frequency or pitch.) The pitch numbers are the values that have to be inserted in the SOUND statement pitch parameter to produce the required note. In the next diagram, the pitch number information has been reproduced in 3 equivalent forms, musical stave, piano keyboard and table. The musical stave has been transposed by one octave for convenience. All are equivalent and you should use the one that you are comfortable with. The pitch numbers are not the actual frequencies of the notes; for example middle C is specified by a pitch number of 53 but has an actual frequency of 261.625 Hz.

Note that there are four integers between each note (row) in the table. In western music an octave is divided into 12 equal semitones. With the sound synthesizer in the BBC micro 4 more intervals are available between each semitone so the smallest interval that can be specified is a quarter semitone. This means that there are 48 intervals available in an octave. In the first part of this chapter we will only be concerned with 'musical' sounds and so we'll stick to values of pitch number given in the table.

Stave diagram (transposed one octave)

Keyboard diagram

	C	C#	D	D#	E	F	F#	G	G#	A	A#	B	OCTAVE NO
NOTE	5	9	13	17	21	25	29	33	37	41	45	49	1
	53	57	61	65	69	73	77	81	85	89	93	97	2
	101	105	109	113	117	121	125	129	133	137	141	145	3
	149	153	157	161	165	169	173	177	181	185	189	193	4
	197	201	205	209	213	217	221	225	229	233	237	241	5
	245	249	253										6

Keyboard labels:

B C Bottom — 1 5

C# D#
9 17

D E F G A B C Middle
13 21 25 33 41 49 53

C# D# F# G# A#
57 65 77 85 93

D E F G A B C Top
61 69 73 81 89 97 101

C# D# F# G# A#
105 113 125 133 141

D E F G A B C 4th octave
109 117 121 129 137 145 149

C# D# F# G# A#
153 161 173 181 189

D E F G A B C 5th octave
157 165 169 177 185 193 197

C# D# F# G# A#
201 209 221 229 237

D E F G A B C
205 213 217 225 233 241 245

C# D
249 253

D, the duration of the sound, is an integer in the range 0 to 254 and specifies a duration value in twentieths of a second. D=254 is the maximum controlled duration and D=255 means that the sound continues indefinitely.

Here, two SOUND statements are shown together with the waveform that they generate.
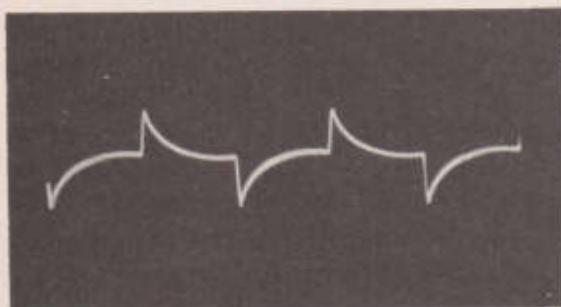
SOUND 1, −15, 100, 255
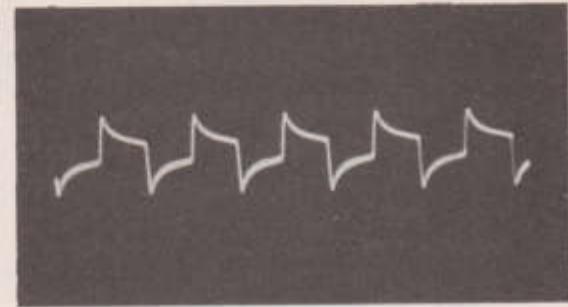
SOUND 1, −15, 200, 255

Using the basic SOUND statement is very easy as the next program demonstrates.

```
10      MODE 1
20      VDU 19, 3, 4; 0;
30      GCOL 0, 1
40      SOUND 1, -12, 53, 80
50      PROCdrawsine(4, 300)
60      GCOL 0, 2
70      SOUND 1, -12, 69, 80
80      PROCdrawsine(5, 200)
90      GCOL 0, 3
100     SOUND 1, -12, 81, 80
110     PROCdrawsine(6, 100)
120     END

130     DEF PROCdrawsine(freq, amp)
140     VDU 29, 0; 500;
150     MOVE 0, 0
160     yscale = amp
180     FOR x = 0 TO 1279 STEP 10
190       y = SIN(RAD(freq*x))
200       DRAW x, y*yscale
210     NEXT x
220     ENDPROC
```

The program generates 3 consecutive notes C, E and G. The frequency relationship of E to C is 5:4 and G to C 3:2. Taken together the 3 notes make up a pleasing harmony known as a major triad. The intervals in pitch between C, E and G form a chord that is pleasant to listen to. The purpose of the program is to demonstrate the effect of the frequency

and duration parameters in the SOUND statement. The program generates the note C and plots a sine wave; then it generates E and a corresponding sine wave, with the frequency increased in the ratio 5:4 relative to that for C; finally it generates G, plotting another sine wave with the frequency increased again in the ratio 3:2 relative to that for C. The sine waves are drawn at different amplitudes to enhance clarity. The sounds are all at the same loudness. The duration of each note should be the time taken for each sine wave to be plotted. A duration of 4 seconds or D = 80 gives a reasonable result.

10.3 Note synchronization

The channel-number parameter can be either a single number in the range 0 to 3 or a 4-digit hexadecimal number, with 3 digits controlling synchronization, flushing and continuation and one digit controlling the channel number. Thus C can be written as:

&HSFN

where the ampersand indicates that the parameter is to be treated as a 4-digit hexadecimal number. The sub-parameters are as follows:

$$C = \&HSFN$$

Sub-parameter	Value	Effect
H	0 or 1	continuation control
S	0 to 3	synchronization control
F	0 or 1	flush control
N	0 to 3	channel number

Thus in the SOUND statement used above, where only a channel number was specified, the other controls would all be set to zero. An explanation of continuation control is postponed until the ENVELOPE statement is covered. How the other controls work is now described.

Flush control

Let us now revisit the three note major triad graphics program. In this program we estimated the duration of each note at 4 seconds or D = 80. However, see what happens if you make D = 200. Here there is no synchronization between the sound and the graph. This is because the second SOUND statement is encountered by the computer before the first has finished. Similarly the third is encountered before the second has finished. In this situation the second and third sounds are placed in a queue (because they cannot be sounded

at the instant the SOUND statements are executed) and the computer carries on to draw the next sine wave. Each sound continues for its specified duration.

In general it would be more convenient to make the program initiate each new note when it is ready to start plotting each sine wave. This can be accomplished by using the flush control. If F is zero, then any sound already being played continues for its duration as above. However if we make F = 1, the queue is flushed or emptied and the new sound is initiated immediately the SOUND statement is executed. Thus the notes produced in the above program by the statements

```
40      SOUND 1, -12, 53, 80
70      SOUND 1, -12, 69, 80
100     SOUND 1, -12, 81, 80
```

are approximately synchronized from guesswork. If we lengthen the notes by making the changes

```
40      SOUND 1, -12, 53, 200
70      SOUND 1, -12, 69, 200
100     SOUND 1, -12, 81, 200
```

the result is completely unsynchronized, whereas if we use

```
40      SOUND      1, -12, 53, 200
70      SOUND &0011, -12, 69, 200
100     SOUND &0011, -12, 81, 200
```

it is exactly synchronized by the use of flush control. All we have done is to make the duration much greater than the graph-plotting time and set F = 1 in the second and third SOUND statements. When line 70 is executed the sound generated by line 40 is immediately terminated. Similarly when line 100 is executed the sound generated by line 70 is immediately terminated. Each sound is initiated by the program as soon as it is ready to plot the corresponding sine wave.

Multi-note sounds and flushing

Let us now alter the specification of the major triad/graphics program so that it generates C, then E with C continuing, then G with E continuing. This is perhaps a more realistic tie up with the graphics display because each sine wave remains on the screen after it is plotted. We can accomplish this by simply using three separate channels and

ensuring that each sound continues after initiation of a new
sound on another channel by setting D = 254, the maximum
duration. The sound statements become:

```
 40    SOUND 1, -12, 53, 254
 70    SOUND 2, -12, 69, 254
100    SOUND 3, -12, 81, 254
```

This, however, has the effect that each sound continues for
the same duration and there is no synchronized ending. Also
the maximum duration D = 254 is not long enough for the
complete process and C cuts out before the third sine wave
is drawn. We can overcome all these problems by setting each
note to be everlasting (D = 255) and getting the computer to
stop all the notes at the same time — when the plotting
process is complete.This can be accomplished by adding the
following three SOUND statements at the end of the program.

```
111    SOUND &0011, 0, 0, 0
112    SOUND &0012, 0, 0, 0
113    SOUND &0013, 0, 0, 0
```

Each channel is flushed immediately by a sound of 0
amplitude and minimum duration. Incidentally we could have
stopped the notes together, without using the flush control,
by using TIME:

```
 39    TIME = 0
 40    SOUND 1, -12, 53, 254
 70    SOUND 2, -12, 69, 254 - TIME/5
100    SOUND 3, -12, 81, 254 - TIME/5
```

but this has the disadvantage that it doesn't necessarily
stop when the plot process is finished but stops after
254/20 s.

Multi-notes and synchronization
To generate a chord with all three notes <u>starting</u>
<u>simultaneously</u> we can use:

```
10    SOUND 1, -10, 53, 40
20    SOUND 2, -10, 69, 40
30    SOUND 3, -10, 81, 40
```

This will produce the C major triad, but with E starting

very slightly after C, and G very slightly after E because
of the interpret time for these three consecutive
statements. Using the synchronization control we can
guarantee that the three notes start simultaneously:

```
10    SOUND &0201, -10, 53, 40
20    SOUND &0202, -10, 69, 40
30    SOUND &0203, -10, 81, 40
```

Actually although it is always claimed that the synchronized
chord sounds 'cleaner' or less 'fuzzy' than the
unsynchronized chord, in the authors' opinion you would have
to be a musical genius to notice the difference. The real
point of the synchronization facility is that we can obey
other statements between SOUND statements and still achieve
synchronization. For example,

```
10    SOUND &0201, -10, 53, 40
15    FOR i = 1 TO 1000 : NEXT i
20    SOUND &0202, -10, 69, 40
30    SOUND &0203, -10, 81, 40
```

produces a perfect chord at the time line 30 is executed.

Now in the above the synchronization control was set to 2
for each SOUND statement. This means that no sound is to be
started until 2 other notes with the same synchronization
control are ready to play. When the first SOUND statement is
encountered nothing happens. Similarly when the second SOUND
statement is encountered nothing happens. It is only when
the third is interpreted that all three sounds are played.
The effect of the synchronization parameter is:

value of S	effect
0	Each sound is inserted in the channel queue
1	The sound is delayed until 1 other is ready from another channel
2	The sound is delayed until 2 others are ready from 2 other channels
3	The sound is delayed until all channels have a sound ready

Now if we return to the flushing statements that terminated
the chord at the end of the previous section,

```
111    SOUND &0011, 0, 0, 0
112    SOUND &0012, 0, 0, 0
113    SOUND &0013, 0, 0, 0
```

we can now see that strictly speaking these three blank
sounds should be synchronized. (Although as already
mentioned the practical difference is imperceptible.)

```
111    SOUND &0212, 0, 0, 0
112    SOUND &0212, 0, 0, 0
113    SOUND &0213, 0, 0, 0
```

10.4 Playing sequences of notes

There are three basic methods that can be employed to get
your computer to play tunes. Firstly the tune can be
incorporated in the program note by note as a DATA
statement, with a pitch, duration and amplitude value for
each note. Secondly you can play a tune by 'typing' keys on
the keyboard, as if it were a keyboard instrument, and
thirdly you can get the computer to make up tunes. The first
method is rather tedious, the second suffers from practical
disadvantages outlined below. The third method is
potentially the most interesting and rewarding. It is
necessary before moving on to the third method to develop
certain techniques and these are most easily demonstrated by
the first two methods: getting the computer to play DATA
statements or playing a tune from the keyboard. Now all of
the techniques described in this section can be enhanced
musically by the use of the ENVELOPE statement so this
section and the material on the ENVELOPE statement should be
read in parallel; for clarity we have kept the material on
generating sequences of notes separate from that on using
ENVELOPE statements for modifying the sound of notes.

Generating scales

When you move on to using the computer as a composer then
you will need a basis or a vocabulary for the music that you
are going to generate. Unless your tastes incline towards
the avant garde, then you will probably want to generate
music in a particular style or feel. You might want to
generate music with a 'folksy' feel or a jazz feel. One
important basis or vocabulary that can be used is scales. A
scale is also an easily generated fixed sequence of notes
and it seems useful to start with scales. They do make a
somewhat boring sound but bear in mind their potential
usefulness when you work through this section.

A scale is a sequence of notes spanning an octave, i.e. a
sequence terminating on the note an octave higher than the
one it started on. The 'do, re, me,...' scale is a major

scale and in the key of C is:

    C D E F G A B C

A scale has a particular musical character (a minor scale is recognizably different from a major scale) and can be expressed as a series of semitone, whole tone or minor third (3 semitones) intervals. The C major scale intervals are:
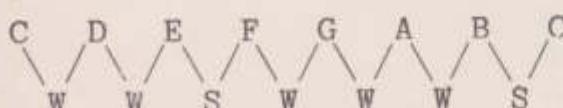
    C   D   E   F   G   A   B   C
     \ / \ / \ / \ / \ / \ / \ /   W = Whole tone
      W   W   S   W   W   W   S   S = Semitone

The major scale can be built on any starting note using the above interval sequence so there are 12 major scales with the interval sequence

    W W S W W W S

It is this interval sequence that imparts the musical character to a scale, not its starting note. A G major scale sounds like a C major scale except for being higher. Different musical forms tend to have notes that are selected from a particular scale. Folk music tends to have notes selected from the pentatonic scale. (Auld Lang Syne is made up entirely of notes from the pentatonic scale.) Jazz blues contain, predominantly, notes selected from a blues scale. A classical work in D-minor contains notes predominantly in the minor scale. It must be emphasized that this is only a first approximation, music is much more complex than this. Paradoxically one of the ways that the listener's interest in a piece of music is held is by creating tension, or selecting notes that do not belong to the underlying scale.

Now because a scale is a sequence of tones and semitones we can play a scale by inserting a SOUND statement in a FOR loop and reading from a DATA statement the tone-semitone sequence. The numbers in the DATA statement are a sequence of 12s, 8s, and 4s. An increment of 8 in the pitch number causes a whole tone interval, an increment of 4 a semitone and an increment of 12 three semitones or a minor third.

Thus a C major scale could be played by:

```
10    tone = 53
20    SOUND 1, -10, tone, 6
30    FOR note = 2 TO 8
40      READ interval
50      tone = tone + interval
60      SOUND 1, -10, tone, 6
70    NEXT note
80    DATA 8, 8, 4, 8, 8, 8, 4
```

The next program expands this idea and selects one out of n scales (3 shown in program). Each scale can be a different length and each DATA statement comprises a key (abbreviated scale name), a length, then the tone semitone sequence.

```
 10    INPUT "Scale", scale$
 20    PROCselectscale(scale$)
 30    PROCplayscale
 40    END

 50    DEF PROCselectscale(scale$)
 70      REPEAT
 80        READ reqscale$
 90      UNTIL reqscale$ = scale$
100    ENDPROC

110    DEF PROCplayscale
120    READ slength
130    note = 53
140    SOUND 1, -10, note, 3
150    FOR i  = 2 TO slength
160      READ interval
170      note = note + interval
180      SOUND 1, -10, note, 3
190    NEXT i

200    DATA wt, 7, 8, 8, 8, 8, 8, 8
210    DATA dim,9, 4, 8, 4, 8, 4, 8, 4, 8
220    DATA maj,8, 8, 8, 4, 8, 8, 8, 4
```

The abbreviations used are wt: whole tone, dim: diminished and maj: major. The following is a list of common scales in the form of pitch number intervals corresponding to tone, semitone and minor third intervals.

Scale	Interval sequence							
major	8	8	4	8	8	8	4	
diminished	4	8	4	8	4	8	4	8
blues	12	8	4	4	12	8		
Hindu	8	8	4	8	4	8	8	
whole tone	8	8	8	8	8	8		
dorian minor	8	4	8	8	8	4	8	
aeolian minor	8	4	8	8	4	8	8	
harmonic minor	8	4	8	8	4	12	4	
pentatonic	8	8	12	8	12			

In the above program PROCselect is a general procedure that could be used in any context where a particular DATA statement is to supply data to a process. Each DATA statement is preceded by a code to be matched with an input

code. PROCselect simply swallows DATA statements until the required one is reached, leaving the READ 'pointer' or 'window' positioned on the length of the required scale. This structure is easily altered to a guess this scale program, selecting at random a DATA statement and checking a guess against the corresponding identification code.

Now all the scales have a different quality that is determined by the interval sequence. For example, the familiar major scale has a suspension on the B that you feel wants to resolve to the C or tonic note. Try playing this scale and stopping at the B.

As the program is written, the different scales all start from the same note – bottom C. Note that the sequences could be built on any starting note:

```
10     INPUT scale$, startnote
20     PROCselectscale(scale$)
30     PROCplayscale(startnote)
         .
         .
130       note = startnote
```

We can thus change key easily. Also note that the speed at which the scales are played and the loudness can be similarly controlled by single parameters.

```
10     INPUT scale$, startnote, notelength, vol
20     PROCselectscale(scale$)
30     PROCplayscale(startnote, notelength, vol)
         .
130     note = startnote
140     SOUND 1, vol, note, notelength
         .
180       SOUND 1, vol, note, notelength
```

Playing tunes from DATA statements

In this section we describe how you program the computer to play tunes from DATA statements. The programming itself is very simple and most of the work involved is in transcribing the music into a form suitable for input to a program.

Now you will never program the computer to produce a really musical performance, but you can have fun rearranging your favourite tunes and adding effects. A good analogy is a mechanical piano playing a tune from a paper roll. The performance is constrained by the limitations of the reproducing mechanism and the information on the paper roll. Basic mechanical pianos play music that sounds mechanical. More elaborate mechanical pianos are very good and practically indistinguishable from a live performance.
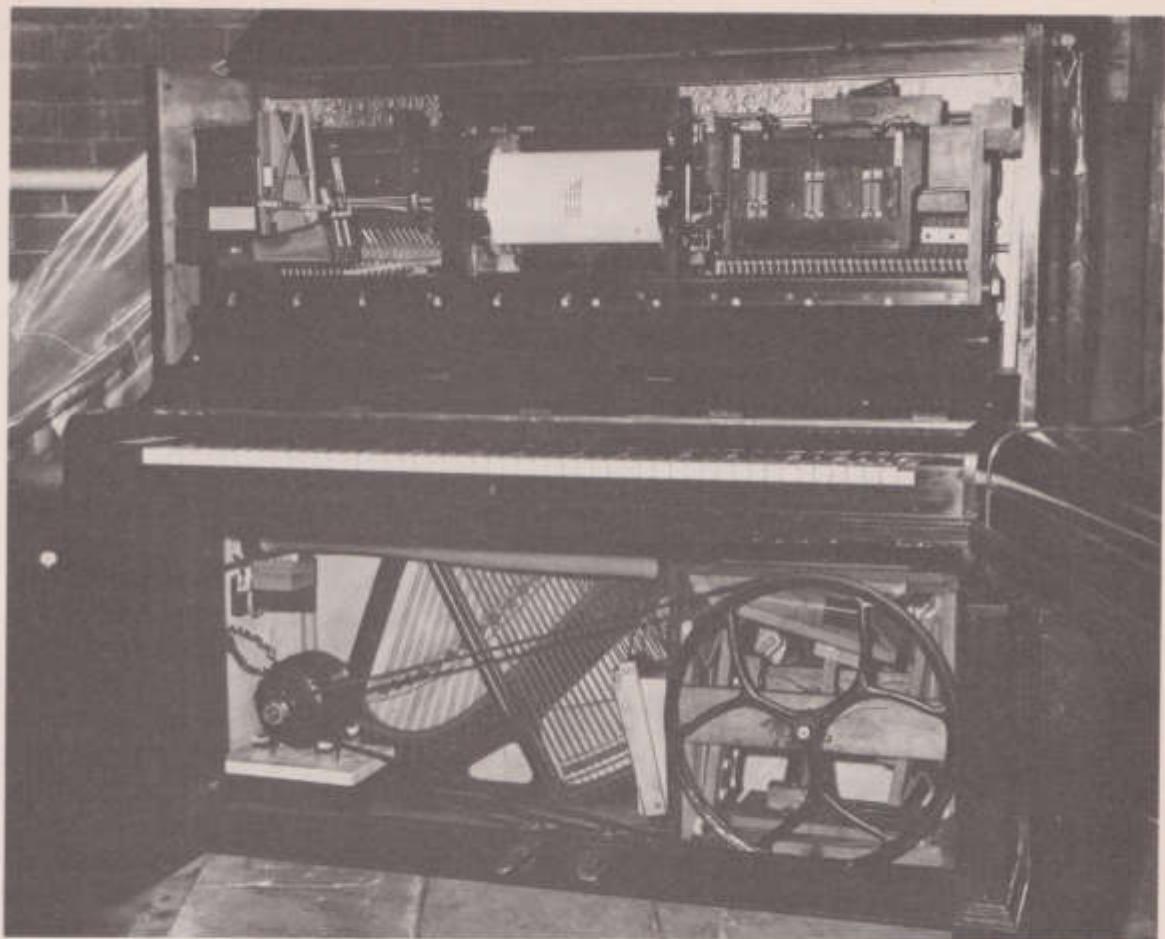
A Hupfield-Phonoliszt reproducing piano, dating from the turn of the century. The paper roll, which has both pitch and expression channels, can be seen above the centre of the keyboard (photo courtesy of The British Piano Museum Charity Trust, Brentford, Middlesex.)

88 pitch channels

Expression channels

Paper roll

Analogy between SOUND statement and a paper roll for a reproducing piano:
Pitch = channel (1 out of 88)
Duration = length of perforation
Loudness controlled by 'extra' expression channels
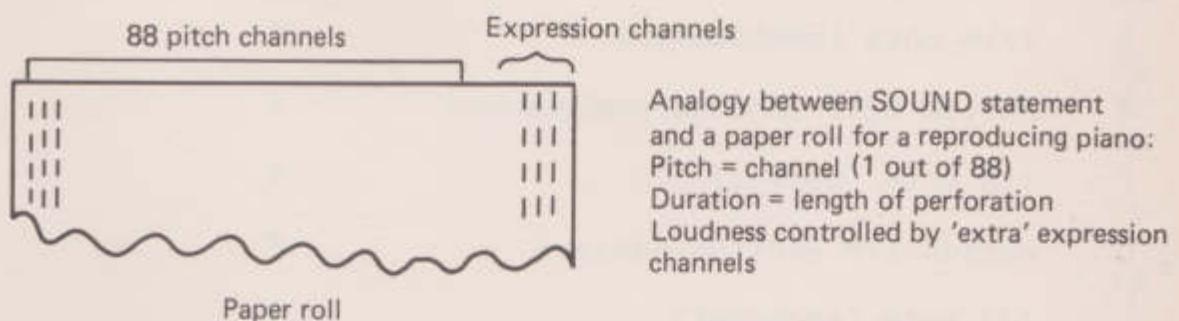
The point of this diversion is that the very elaborate reproducing pianos have much more information on the paper roll than do the basic ones. The same principle applies to the tune DATA statements - the more expression and variety that you wish to build into the tune, the more information must be loaded into the DATA stament for each note.

Certain alterations have to be made to the scale-playing program to enable the computer to play a recognisable tune from DATA statements, rather than just a scale. We must specify, individually for each note, both its pitch number and its duration. Note that a tune can also be stored in a DATA statement as a set of intervals, just as we did for scales. However, transposing a tune into a set of intervals is generally more difficult than transposing it into a set of pitch numbers. The intervals in a tune would of course be both positive and negative.

Later we will consider also specifying the loudness of each individual note. Thus each note in the tune must be specified as two or three data items representing:

(1) pitch number
(2) duration
(3) loudness

Selecting the pitch number and duration is just a matter of transposing the normal musical notation into numbers. Duration can be worked out using any convenient scheme such as:

Musical convention	Duration value (for metronome 150)
1/32 note (demi-semiquaver)	1
1/16 note (semiquaver)	2
dotted 1/16 (dotted semiquaver)	3
1/8 note (quaver)	4
dotted 1/8 (dotted quaver)	6
1/4 note (crotchet)	8
dotted 1/4 (dotted crotchet)	12
1/2 note (minim)	16
dotted 1/2 (dotted minim)	24
whole note (semibreve)	32

Note that there are notes that cannot be accurately represented at this tempo. For example, a dotted 1/32 note is 1.5 units in duration (only integers can be used as duration parameters in SOUND statements). Similarly a 1/16 triplet is 4/3 units per note, an 1/8 triplet 8/3 units per note and a 1/4 triplet 16/3 units per note. Use of these phrases must be avoided or, if you are transposing composed music triplets must be altered. Alternatively, all the times could be multiplied by 3 and the music played at a slower tempo. In the table a crotchet is sustained for 8/20 second. Thus, using the above values for a piece of music would mean that the music was played at a rate of (20x60)/8 crotchets per minute - a metronome tempo of 150. These settings can be used for faster or slower music and they have the advantage that they are all integers. The duration parameter in the SOUND statement is multiplied by a constant that speeds up or slows down the above rate.

```
tempo = 2
    .
    .
    .
SOUND 1, vol, note, tempo*standardvalue
```

or

```
tempo = 0.5   :REM only if there are no odd durations
    .
    .
    .
SOUND 1, vol, note, tempo*standardvalue
```

Here is a program that can be used to play a piece of music, given DATA statements containing pitch and duration codes for each note.

```
  5    INPUT tempo
 10    READ note, duration
 20    REPEAT
 30      SOUND 1, -10, note, tempo*duration
 40      SOUND 1,   0, 0, tempo*2
 50      READ note, duration
 60    UNTIL note = -1

 70    DATA 69,8, 81,16, ...
       ... pitch and duration codes  ...
100    DATA -1,-1
```

Another point to note is that we have included a gap of silence between each note (line 40). In music sometimes notes are slurred together, sometimes not - it's a matter for experimentation. This is not an entirely satisfactory solution since it will disrupt the phrasing of the music,

i.e. the delays inserted in each bar will depend on the number of notes in the bar. The use of the amplitude ENVELOPE control (later) eliminates the need for the gap.

The pitch and duration values for the traditional ballad "Greensleeves" are now given, along with the music.

Greensleeves   (Traditional)
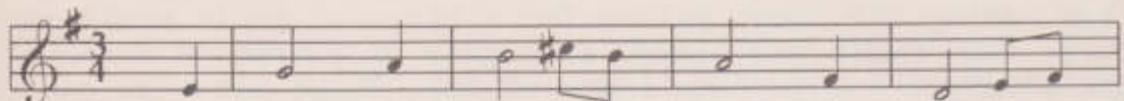
Pitch	69	81	89	97	105	97	89	77	61	69	77
Duration	8	16	8	16	4	4	16	8	16	4	4

Pitch	81	69	69	65	69	77	65	49	69
Duration	16	8	12	4	8	16	8	16	8

Pitch	81	89	97	105	97	89	77	61	69	77
Duration	16	8	16	4	4	16	8	16	4	4
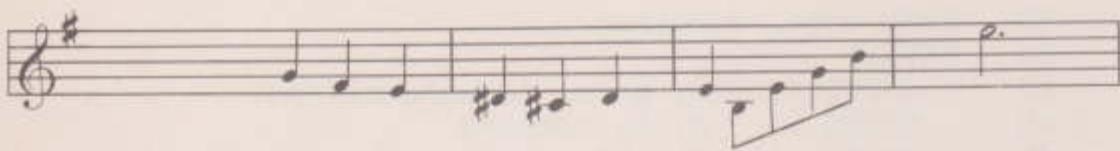
Pitch	81	77	69	65	57	65	69	49	69	81	97	117
Duration	8	8	8	8	8	8	8	4	4	4	4	16

More complicated rhythm and rests

Interesting music contains spaces or gaps of silence that have rhythmic significance. These are notated by rest marks. Again using the metronome tempo 150, the durations for each rest mark are as follows:

Musical convention	Duration (for metronome 150)
♬ 1/32 (demi-semiquaver) rest	1
♬ 1/16 (semiquaver) rest	2
♪ 1/8 (quaver) rest	4
♪ 1/4 (crotchet) rest	8
▬ 1/2 (minim) rest	16
▬ whole (semibreve) rest	32

In order to handle a piece of music that is transposed for rest marks and notes, we need a switch in the program that will enable notes or silence to be played for the appropriate time. To save having three DATA items per note we could use the highest pitch number to indicate that a rest is to be 'played'. Thus a rest appears in the DATA statement as a note of pitch number 255. This note is not now available and the rest is played at this pitch but with zero loudness.

```
10    firstime = TRUE
20    INPUT tempo
30    READ note, duration
40    REPEAT
50      IF note = 255 THEN loudness = 0
        ELSE                loudness = -10
60      IF firstime AND note = 0 THEN
            RESTORE : firstime = FALSE
70      SOUND 1, loudness, note, tempo*duration
80      SOUND 1, 0, note, tempo * 2
90      READ note, duration
100   UNTIL note = -1

110   DATA 255, 4, 53, 4, ...
          .
          .
          .
120   DATA 0, 0   :REM indicates repeat from start
          .
          .
          .
200   DATA -1, -1
```

We have also included a structure that deals with a common situation in themes – repeating a sequence of bars. In the above theme the first four bars are to be repeated. This is accomplished by using an IF statement in conjunction

232

with a RESTORE. We now need another data terminator at the end of the first four bars and have used the lowest pitch number – zero.

Here is a piece of music that includes rests and a repeat marker.

Tomboy   (Steve Salfield)

Pitch	255	53	255	101	255	53	85	81	73	65	53	33	53	255	65	73	53
Duration	4	4	12	4	4	4	4	4	4	8	8	4	8	8	4	8	36
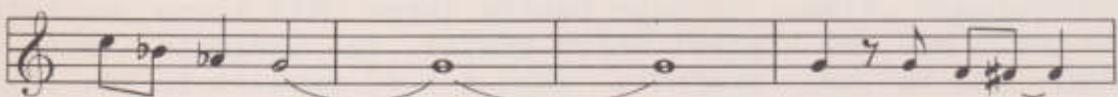
Pitch	101	93	85	81	81	255	81	73	77
Duration	4	4	8	80	8	4	4	4	12
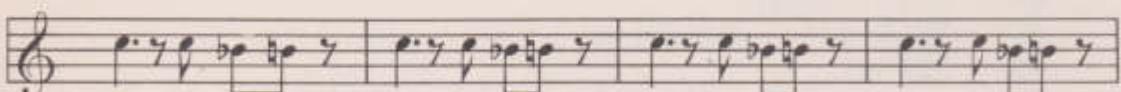
Pitch	101	255	101	93	97	255
Duration	12	4	4	4	4	4

Having explored the techniques to play single-note melodies, the two other channels can be used for chords and counter melodies. This means adding more information into the DATA statements and its success depends on your skill as an arranger! Such additions do not require any further programming tools (although a little knowledge of music is perhaps necessary) and have been left as an exercise for the reader. Later we will be returning to these tunes to add rhythmic and other effects.

Exercises

1 Alter the "Greensleeves" program so that each note has three variable parameters and three items in the DATA statement. The third item is to be loudness. Try accenting the first note in each bar.

2 Write a program to change the pitch numbers in the DATA statement so that intervals are defined rather than pitch numbers. The key of "Greensleeves" is G major, so the first interval is 9 semitones or 36. All other intervals are then derived from the pitch numbers by subtracting the (n-1)th. from the nth.

1st interval	36
2nd interval	81-69
3rd interval	89-81
4th interval	97-89
⋮	

The tune can then be played using the incremental technique used for scales, in any key. Use file facilities or an array to store the interval sequence.

10.5 Playing tunes from the keyboard

At this stage it is useful to introduce a program that will play notes from the keyboard and also change the characteristics of the note played as soon as an 'effect' key or 'organ stop' is pressed. You can then play tunes from the keyboard and instantly change the sound produced by pressing an 'effect' key, just as you would on an electronic organ. The structure of the program is amenable to any extensions that you wish to experiment with.

The program causes the keys on the bottom row of the keyboard to behave like the white notes on an organ or piano, with the keys on the row above behaving like black notes where appropriate. The key C represents Middle C. The schemes that you can set up are not limited to just playing notes. Keys can also be used to initiate effects as in an electronic organ. A key could control harmony, for example, so that when depressed, two or three SOUND statements play the note indicated by the pitch key. The SOUND statements could be harmonised in octaves, major fourths, etc. A key could control effects and the selection of ENVELOPE statements to control the SOUND statement (below).

Let us consider the basic keying program that has to play a SOUND statement for as long as the key is depressed.

```
10    keys$ = "ZSXCFVGBNJMK,L./: "
20    *FX 11, 1
30    *FX 12, 1

40    currnote$=GET$
50    REPEAT
60      pitch = INSTR(keys$, currnote$)*4 + 37
70      SOUND 1, -15, pitch, 255
80      REPEAT
90        note$ = INKEY$(2)
100     UNTIL note$<>currnote$
110     SOUND &11, 0, 0, 0
120     IF note$ = "" THEN currnote$ = GET$
                        ELSE currnote$ = note$
130   UNTIL currnote$ = " "

140   *FX 12,0
150   END
```

In the program the inner REPEAT loop causes the SOUND statement to be played UNTIL the key is no longer pressed or UNTIL a different key is pressed. The INKEY parameter defines the length of time that the program is to wait and see if a key is pressed. The value of this parameter has to be as small as possible (so that playing the keys fast is possible), but greater than the repeat rate at which characters are sent by a continually depressed key. The repeat rate is decreased to its smallest possible value using a *FX call (see Appendix 10).

Handling effects
The above structure can easily be enhanced to include special effect keys. At this stage we introduce the use of the ENVELOPE statement which modifies the generated sound. An explanation of the ENVELOPE statement is postponed until later. For the moment all you need know is that an integer in the range 1-4 in the second (loudness) parameter in the SOUND statement refers to a predefined ENVELOPE statement. The 'effect' keys 1, 2, 3, 4 select one of four envelopes for playing subsequent notes and the P key (P for Plain) causes subsequent notes to be played without an envelope. The O key (O for Octave) causes each note to be played together with the note an octave above (using the same envelope for both notes). The effect keys are checked by PROCcheckforstops, in between each note being played. A variety of stops could be examined by extending this procedure. The procedure also has to absorb the variable number of characters that will be sent by a stop key due to the increased key repeat rate.

You can define up to four ENVELOPE statements and these can be incorporated into a number of effect keys. Some suggestions for further extensions are:

(1) Single notes with a selected ENVELOPE (4 possibilities)

(2) Chords can be played with 2 or 3 SOUND statements selecting one of the 4 ENVELOPE statements (8 possibilities) for each note in the chord.

(3) Playing chords where each note in the chord is controlled by a different ENVELOPE statement (4x3x2 possibilities per 3 note chord)

(4) Adding Channel 0 effects.

You can see from this cursory list that you will run out of keys quicker than effects.

```
1       ENVELOPE 1,1, 0,0,0,0,0,0,
                      126,-8,0,0,126,50
2       ENVELOPE 2,1, 0,0,0,0,0,0,
                      126,-8,-4,-1,126,100
3       ENVELOPE 3,1, 0,0,0,0,0,0,
                      63,10,0,-63,63,126
4       ENVELOPE 4,1, 0,0,0,0,0,0,
                      32,63,0,-63,63,126

10      octave = FALSE : envelope = -15
20      keys$ = "ZSXCFVGBNJMK,L./: "
30      *FX 11, 1
40      *FX 12, 1

50      currnote$=GET$
60      REPEAT
70        PROCcheckforstops
80        pitch = INSTR(keys$, currnote$)*4 + 37
90        SOUND 1,envelope,pitch,255
100       IF octave THEN SOUND 2,envelope,pitch+48,255
110       REPEAT
120         note$ = INKEY$(2)
130       UNTIL note$<>currnote$
140       SOUND &11, 0, 0, 0
150       SOUND &12, 0, 0, 0
160       IF note$ = "" THEN currnote$ = GET$
                        ELSE currnote$ = note$
170     UNTIL currnote$ = " "

180     *FX 12,0
190     END
```

```
210    DEF PROCcheckforstops
220       IF INSTR(keys$, currnote$)>0 THEN ENDPROC
230       IF currnote$ = "O" THEN octave = NOT octave
240       IF currnote$ = "P" THEN envelope = -15
250       IF INSTR("1234", currnote$) THEN
                envelope = ASC(currnote$)-ASC("0")
260       REPEAT: currnote$ = GET$
270       UNTIL INSTR(keys$, currnote$)
280    ENDPROC
```

10.6 Composing music with programs

There are many different ways in which a program can be made
to generate music (perhaps 'generate' is a better word than
'compose' in the context of computer programs). One of the
most common was used by Mozart when he invented a game that
composed music using a pair of dice. The computer equivalent
is to use a random number generator to generate musical
patterns. The simplest scheme would involve the generation
of a sequence of notes of random pitch and random duration
but a melody generated by such a method might not be very
interesting.

At the other end of the spectrum a melody generating
program can be written to produce a particular musical style
evocative of a certain musical genre or even a certain
composer. Here considerable information is required by the
program to categorize the style to be imitated. This may be
in the form of probability distributions, not just first-
order distributions (the probability that a certain note
will occur), but second and higher-order distributions. A
second-order probability distribution gives the probability
that a certain note, x, will occur in a sequence, given that
it is preceded by a note y. This makes the selection of a
note dependent on the note(s) that have preceded it.

A general principle is that the more closely a program is
required to approximate a style, the more information
(probability distributions and other constraints) it needs
about that style.

Because of their general importance, we shall first of
all look at the technique of using first-order probability
distributions.

Use of probability distributions in random selection

This technique is quite general and could be used in any
context, computer poetry writing for example, where a random
outcome is to be weighted according to a probability
distribution. It's rather like a weighted dice. We do not
want the dice to fall in such a way that there is a 1 in 6
chance of a particular face turning up. A weight inserted in
the dice will influence its random behaviour and will cause
one or more faces to be favoured. In a program the random

number generator is the dice, and the weight is the
probability distribution. For example, let us say that we
want to generate a melody by selecting notes from the scale
of C major, the probability of occurrence for each note
being as follows:

Note	probability of occurrence
C (bottom)	13%
D	13%
E	15%
F	10%
G	16%
A	9%
B	12%
C (top)	12%

These values might have been generated as a result of an
analysis of a selection of music in the style in which we
are interested.

We now explain how to generate a note so that the
probability of getting a particular note matches its entry
in the above table. The basic technique used can be extended
to deal with higher order probability distributions which
are beyond the scope of this book.

We wish to generate a random number in the range 1 to 100
and select a note as follows:

Random number generated	Note chosen
1 - 13	C
14 - 26	D
27 - 41	E
42 - 51	F
52 - 67	G
68 - 76	A
77 - 88	B
89 - 100	C

We first set up an array containing the eight
percentages. In order to select a note, we generate a random
number in the range 1 to 100 and add up values from the
array until the total is greater than or equal to the random
number generated. The number of percentages added determine
the note from the scale that is selected.

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
13	13	15	10	16	9	12	12

rand=RND(100)   Say, for example, rand=43.   Then the
                     first 4 values in the frequency table
                     are added before we obtain a total
                     that exceeds 43.   This means that we
                     play the 4th note in the scale.

pitch=scalenote(4)

                     where the 8 pitch codes for our scale
                     are held in an array 'scalenote'.

First-order   probability   distributions   are   not
particularly   useful   in   computer music generation or style
synthesis -   intervals   are   far   more   important   than   the
absolute   pitch   of a note. That is, given that a particular
note   has   occurred   the   interval to   the   next   note   is
important.   If   you   use   probability distributions in music
generation   you   must   use   second-order and   above.   The
mathematics   of   second   order   probabilities is outside the
scope of this   text.   First-order   probabilities   have   been
dealt   with because of their general importance in different
contexts requiring weighted   random   numbers.   Try   a   music
generating   program   using   the   above technique and see how
inferior it is to easier ad hoc techniques now introduced.

A practical program to generate music in a particular style
We shall now   proceed   to   develop   a   program,   more   on   a
pragmatic   or ad hoc basis than on a mathematical basis. The
program is the basic structure   for   a   musical   composition
program   and   can   easily   be   extended   to incorporate more
variety, etc., as suggested.
     This program   uses,   as   a   musical   vocabulary   or   note
source,   a   jazz blues scale given as absolute pitch numbers
on line 50. The program plays a 12 bar blues in   this   scale
by:

(1) randomly selecting a starting note,

(2) randomly selecting a rhythm for a musical phrase, and,

(3) randomly   selecting   whether   the phrase is to ascend or
    descend in major seconds.

```
10      DIM blue(13)
20      FOR note=1 TO 13
30        READ blue(note)
40      NEXT note
50      DATA 45,57,65,69,73,85,93,105,113,117,121,133,141
60      FOR bar=1 TO 12
70        FORphrase=1 TO 2
80          PROCselectstartnote
90          PROCselectupdown
100         PROCselectphrase
110         PROCplayphrase
120       NEXT phrase
130     NEXT bar
140     END

150     DEF PROCselectstartnote
160        startnote = RND(13)
170     ENDPROC

180     DEF PROCselectphrase
190        sphrase=RND(5)
200        restoreto = 1000 +sphrase
210     RESTORE restoreto
220     ENDPROC

230     DEF PROCplayphrase
240       READ noofnotes
250       note = startnote
260       FOR i = 1 TO noofnotes
270         READ duration
280         SOUND 1,-10,blue(note),duration
290         note=note+ updown
300         IF note > 13 OR note < 1 THEN note=7
310       NEXT i
330     ENDPROC

340     DEF PROCselectupdown
350        IF RND(2) = 2 THEN updown= -1  ELSE updown=  1
360     ENDPROC

1001    DATA 8,2,2,2,2,2,2,2,2
1002    DATA 4,4,4,4,4
1003    DATA 2,8,8
1004    DATA 1,16
1005    DATA 4,2,6,2,6
```

The pitch numbers are stored in array 'blue'. The procedure
that selects a start note is self-explanatory. Each phrase
lasts for 1/2 note (duration = 16) and the procedure that
selects the phrase selects a duration DATA statement by
RESTOREing under control of a random number generator. (Note
that this facility is sensitive to line renumbering.)

PROCselectupdown controls whether the phrase is to ascend or descend and is self-explanatory. Line 300 arbitrarily sets the melody back to the 7th note in the scale if the generated sequence goes out of range.

Now this structure can easily be built upon by both constraining and extending the musical devices used; the extensions add variety and the constraints add realism. The suggestions are fairly arbitrary and you can make up your own devices and add effects as outlined in the next section.

Suggested extensions

(1) Extend rhythm phrases.

(2) Introduce rests in phrases.

(3) Allow intervals other than major seconds (i.e. allow 'note' to be incremented and decremented by an integer other than 1).

(4) Musically improve the 'out of range' recovery.

Suggested constraints

(1) Fast note phrases used consecutively should be followed by a long note.

(2) The final note should be the tonic (1st, 7th, or 13th) and should be a long note.

(3) A short note ascending passage should be followed by a descending passage or interval.

(4) Repetition of a phrase should be occasionally introduced.

(5) There are harmonic consraints given by the blues chord progression that can be used if you know the musical theory.

Eventually you should end up with a recognisably 'bluesy' sound that can then be elaborated with rhythm and effects covered in the next section.

10.7 The ENVELOPE statement: amplitude

The sounds produced by different types of musical instruments are different. Indeed the sounds produced by one instrument played by different players is different. These are facts that are familiar to everyone that has listened to music. These differences can be quantified and two mathematical models, or ways of quantifying the differences heard by the human ear, are applied to musical sounds.

Firstly consider a note played on an instrument and held at a steady volume. A continuous note played on a trumpet sounds totally different from the same continuous note played on an oboe. Why is this so? In the simplest terms it is because the mechanical design of an instrument causes a combination of vibration modes at different frequencies and the differences in mechanical design mean that different instruments exhibit varying vibrational strengths at different modes. A continuous note played on any instrument can be considered to consist of a series of superimposed sine waves all having different frequencies and amplitudes. This is sometimes called a harmonic envelope. The harmonic envelope for a clarinet is shown.

Frequency (Hz)
Harmonic envelope of clarinet playing
a continuous note of 523 Hz
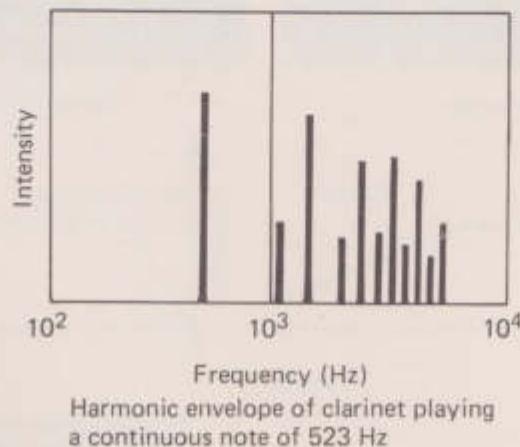
The diagram indicates that this note is made up of a sine wave undulating at 523 Hz (the frequency of top C) plus a number of other sine waves or harmonics undulating at 2 times, 3 times, 4 times etc. the fundamental frequency of 523 Hz. The fundamental sine wave is the one with the highest amplitude, and the relative amplitude of the other harmonics defines a harmonic envelope. The overall shape of this envelope gives a note a characteristic sound, and this shape varies generally with the frequency of the note and the way an instrument is played. Even the same note on the same instrument can produce different envelopes depending on the way it is blown. Now different instruments, due to their different mechanical design, produce completely different

harmonic envelopes. Computer synthesis of continuous tones of different instruments can be achieved by playing pure sine waves of different amplitude and frequency simultaneously. The capacity for the BBC computer to do this is, however, limited by the fact that it has only three tone channels, and besides, the tone generated is not a pure sine wave, but a square wave, already possessing its own harmonic envelope.

There is, however, another important consideration in the sound that a particular instrument makes that the BBC computer can synthesise accurately. This is the amplitude envelope. Above we considered a continuous note, but musical notes are not continuous. They start and stop, some more abruptly than others. They take a certain time to build up and decay. These times are characteristic of different instruments.
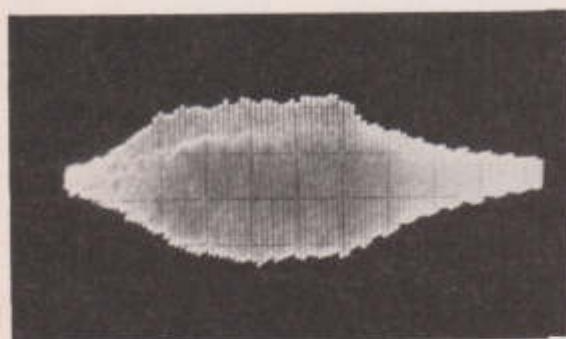
Flute: isolated note
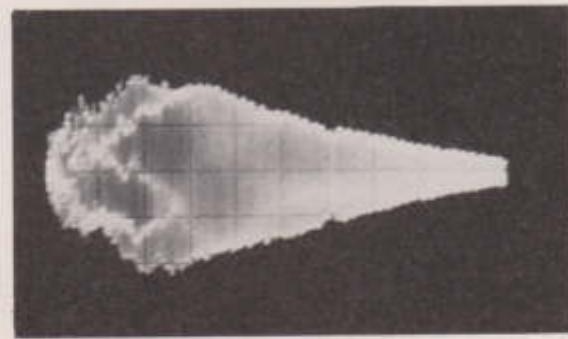
Vibraphone: isolated note
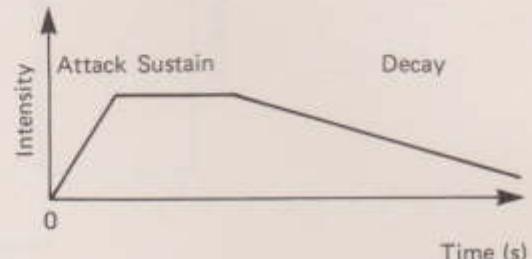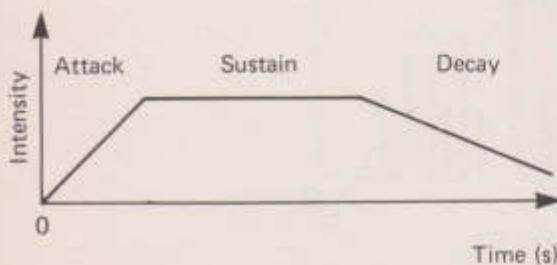
Piecewise-linear approximations
to instrumental transients

The illustration shows one of a series of isolated notes at the same frequency played on a flute, together with one of a series of notes played on a vibraphone (a tuned percussion instrument like a xylophone with metal bars). The waveforms show that the undulations that the instruments make (which themselves can be synthesised from a number of pure tones) have their amplitude controlled by the transient characteristic of the instrument. The amplitude of the undulations build up, sustain and decay. The attack time on the flute, the rate at which the sound builds up to a maximum, is much longer than the attack time for the vibraphone. The sustain time of the flute is longer than that of the vibraphone. (Although sustain time in a wind instrument depends on the player, the illustration above shows the sound produced by a player instructed to play as

short a note as possible). A piano - essentially a percussive instrument, has a transient that exhibits a fast build-up and long sustain. This is almost the opposite of an organ where the note takes a longer time to build up but decays sharply as the air supply is abruptly cut off. In fact a tape recording of a piano, if played backwards, makes organ sounds, and vice versa.

The rate at which a note builds up and decays and how long it is sustained for can be controlled accurately by the ENVELOPE statement, and this is now dealt with in detail. Bear in mind that the undulations whose amplitude is controlled are square waves and not pure tones or 'trumpet-like' or 'clarinet-like' undulations. Approximate instrumental synthesis is still possible, however, and this demonstrates that transient control or control over the amplitude envelope, is just as important as (if not more important than) the harmonic envelope in determining the unique sound of an instrument.

The amplitude ENVELOPE parameters

When a positive integer in the range 1 to 4 is used as a 'loudness' parameter in a SOUND statement, this means that the loudness and variation in pitch of the sound will be controlled by an associated ENVELOPE statement. The amplitude specification part of the ENVELOPE statement controls the transient amplitude of the basic tone waveform, just as instrumental design and player's technique control the transient amplitude of the above waveforms. The amplitude is controlled by the programmer specifying, in the ENVELOPE statement, the shape of a piecewise linear amplitude envelope. The second and last six parameters in the ENVELOPE statement contain this information. Parameters three to eight specify the pitch envelope (see below). These parameters will be set to zero for the time being.

```
ENVELOPE  n, t,
          pi1, pi2, pi3, pn1, pn2, pn3,
          aa, ad, as, ar, ala, ald
```

The first parameter, 'n', is the ENVELOPE number. This is the number referred to in the second parameter of a SOUND statement. Parameter 't' specifies the time increment or step (in 1/100ths of a second) that is subsequently employed by the other parameters. The last six parameters specify the shape of the amplitude envelope as follows:
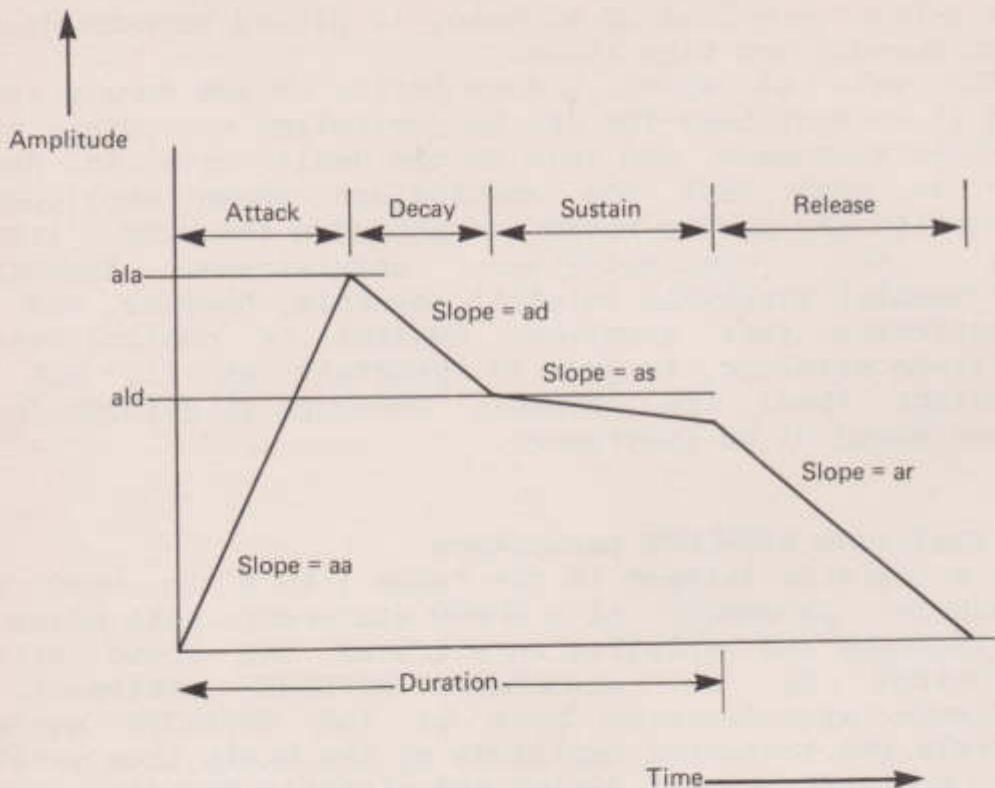
General piecewise-linear form
of an amplitude envelope (t = 1)
The straight lines are
incremental steps.
For example
ala = 120
aa = 30
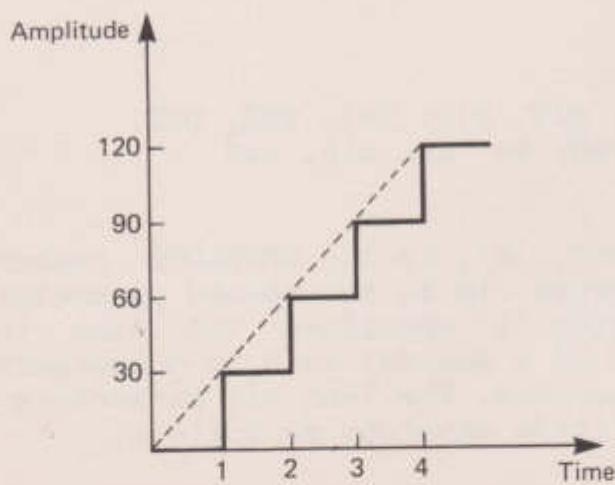
Attack phase

The attack phase of an amplitude envelope determines how long the note takes to build up to a maximum value. This is a very important parameter in determining the overall sound of a note. Percussive instruments, where a resonating body is actually struck by a 'hammer', have very short attack transients. Such instruments are pianos, drums, cymbals, etc. The attack phase is quantified by a slope and a target maximum level. The first parameter, 'aa', specifies the slope of the attack phase and is the change in amplitude per time step 't'. The value of 'aa' lies in the range -127 to 127 and is normally positive. This is used in conjunction with 'ala' the target amplitude at the end of the attack phase. 'ala' takes values in the range 0 to 126 corresponding to amplitude levels in the range 0 to 15 in the SOUND statement. Thus the following values would generate an attack going from 0 to maximum volume in 4/100ths of a second:

```
t    =  0
ala  = 126
aa   = 32
```

An attack reaching a maximum loudness of half full volume in the same time could be achieved by halving both 'ala' and 'aa'.

Decay phase

The second parameter, 'ad', the change of amplitude per time step is similarly used in conjunction with 'ald', the target level at the end of the decay phase. In a real waveform there is no piecewise linearity but a smooth transition from attack to decay to sustain. In fact the decay and sustain are just one decreasing phase where the initial rate of decrease is high. The sound level on stringed instruments dies away rapidly after the string is plucked or struck. Using a piecewise-linear envelope we are able to simulate this effect with straight lines that require only two parameters. Now 'ad' takes values in the range -127 to 127 and is normally negative.

Sustain phase

As we have discussed, notes die away but they do so more quickly at first. We arbitrarily divide the decrease in volume into two parts the fast decrease phase - decay, and the slow decrease phase - sustain. The third parameter, 'ar', takes values -127 to 0 and is the change of amplitude per time step, 't', in the sustain phase. There is no target parameter corresponding to 'ar'. The sound is sustained either at a constant level ('as=0') or at a slight decay, until the duration parameter, in the corresponding SOUND

statement, runs out. (Remember that some of this time has already been taken up by the attack and decay phase.)

Release phase
The next parameter, "ar", (-127 to 0) allows a piecewise-linear modelling of the next characteristic part of a sound waveform - a fall-off in loudness level that is generally faster than the decrease in the sustain phase. Again this does not need a target level (the target is always zero).

The release phase in a note is not necessarily sounded. The duration parameter controls the attack, decay and sustain phase. (Also the sustain phase parameters can be such that the loudness is reduced to zero before the time limit set by the duration parameter.) The release phase is entered only if there is no other note waiting in the channel queue (and the volume has not already been reduced to zero). If the release phase is entered, it is immediately truncated by any other note initiated on that channel.

The release phase can be forced by using two SOUND statements. The first references the ENVELOPE statement in the normal way. The second has "H", the continuation control parameter set to 1. This inserts a dummy note in the queue allowing the release phase of the previous note to sound. The amplitude, pitch and duration parameters of this dummy note are ignored. Thus:

```
SOUND 1, 1, 53, 200
SOUND &1001, 0, 0, 0
```

means that the first SOUND statement is to be controlled by ENVELOPE 1, and the second SOUND statement ensures that the release phase of this note is enabled.

Practical use of amplitude ENVELOPE
The next illustration is an oscilloscope photograph of an isolated note generated using:

```
ENVELOPE 1, 1,
          0, 0, 0, 0, 0, 0,
          20, -20, 0, 0, 126, 0
```

This is an inverted "V" envelope taking seven increments to build up and seven to decay, the attack rate being equal and opposite to the decay rate.
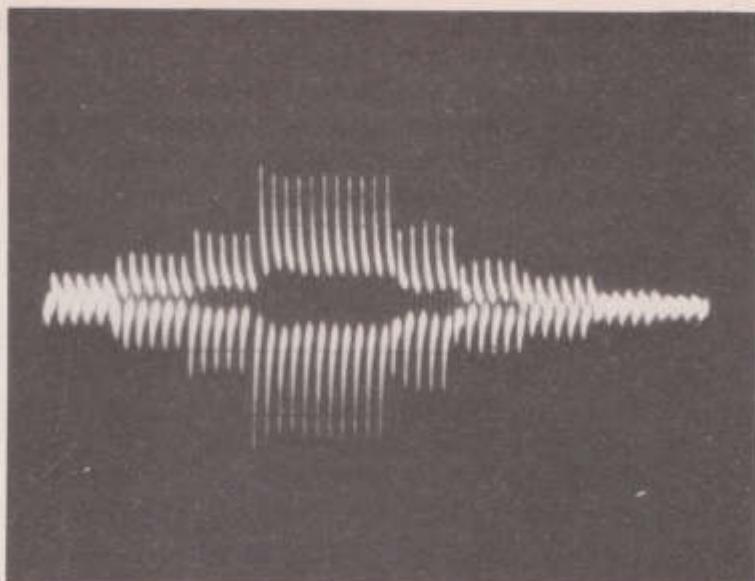
Oscilloscope photograph of
sound under ENVELOPE control
1,1,
0,0,0,0,0,0,
20, −20, 0,0,126,0
(the trace starts on the 3rd increment)
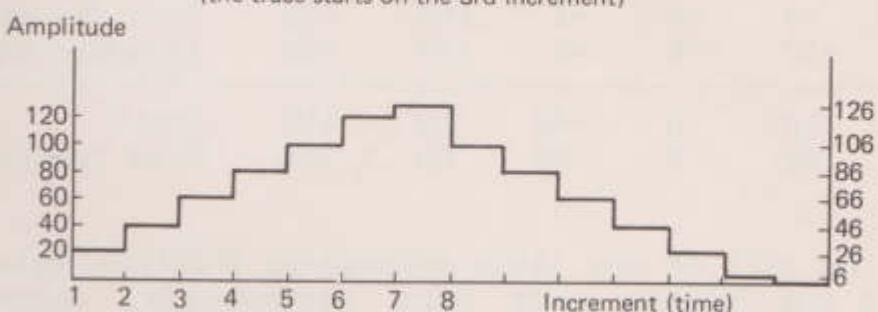
Diagram showing the variation
in amplitude controlled by
the ENVELOPE statement

The next programs allows experimentation with the amplitude
part of the ENVELOPE statement.

```
10      tempo = 1
20      INPUT aa, ad, as, ar, ala, ald
30      ENVELOPE 1, 1,
                0, 0, 0, 0, 0, 0,
                aa, ad, as, ar, ala, ald
40      READ note, duration
50      REPEAT
60        IF note = 255 THEN envelope = 0
                        ELSE envelope = 1
70        SOUND 1, envelope, note, tempo*duration
80        READ note, duration
90      UNTIL note = −1

100     DATA ...
```

Note that we have now omitted the silence gap between notes. As the notes are punctuated with attack, there is now no need for this. Also remember that inter-note gaps tended to upset the musical timing of a piece of music. Another important point concerning the ENVELOPE statement is that it must be obeyed by the program before it is referred to in a SOUND statement. When it is interpreted an ENVELOPE buffer is set up. The effect of having an ENVELOPE statement obeyed after the SOUND statement which refers to it, would either be silence, or whatever previous ENVELOPE has been left in the buffer.

Some amplitude ENVELOPEs are now given together with a very approximate subjective description of their sound or effect. More than anywhere else in the book this is an area for personal experiment. The ENVELOPE parameters are meant to be used with 't=1', as in the program above.

aa	ad	as	ar	ala	ald	
126	-4	0	0	126	100	Electric piano
126	-8	0	0	126	50	Electric guitar
126	-8	-4	-1	126	100	Acoustic piano
126	-10	-5	-2	126	126	Acoustic guitar
63	10	0	-63	63	126	Organ
63	63	0	-63	63	126	Wind instrument

In the top part of the table percussive ENVELOPE values are given, in the bottom part wind instruments, values are suggested. Note that for the wind instruments, values of 'ald' are greater than 'ala', and 'ad' is positive. A long slow attack is continued into the 'decay' phase.

Sound textures using the amplitude ENVELOPE - Echoes
We now move on to consider the effect of using two or three SOUND statements under ENVELOPE control. An echo effect can be achieved by using a second SOUND statement under a different ENVELOPE control, but using the same SOUND channel. A suggestion is:

```
30    ENVELOPE 1, 1,
                0, 0, 0, 0, 0, 0,
                aa, ad, as, ar, ala, ald
35    ENVELOPE 2, 1,
                0, 0, 0, 0, 0, 0,
                aa/2, ad, as, ar, ala/2, ald/2
                      .
                      .
70    SOUND 1, envelope, note, duration * tempo
75    SOUND 1, envelope*2, note, tempo * 2
```

The parameters in the echo ENVELOPE can be made a function of the main envelope parameters to control the echo effect. Again you should experiment.

Sound textures : harmonics

As we discussed earlier, it is not only the transient behaviour of a note that gives it its characteristic instrumental sound, but its harmonics. We can add harmonics to a program-generated note (only two) by playing other notes simultaneously. Also bear in mind that the BBC tones are not sine waves and that is why, no matter how hard you try, all notes will sound 'electronic'. Now, in real harmonic envelopes, not only does the amplitude change as a function of the frequency of the harmonic but so does the attack of the harmonic. We can control this easily in a program

```
30    ENVELOPE 1, 1,
               0, 0, 0, 0, 0, 0,
               aa, ad, as, ar, ala, ald
35    ENVELOPE 2, 1,
               0, 0, 0, 0, 0, 0,
               aa/2, ad, as, ar, ala/2, ald/2
36    ENVELOPE 3, 1,
               0, 0, 0, 0, 0, 0,
               aa/3, ad, as, ar, ala/3, ald/3
               .

               .
70    SOUND 1, envelope, note, tempo * duration
75    SOUND 2, envelope*2, note + 48, tempo * duration
76    SOUND 3, envelope*3, note + 96, tempo * duration
```

The envelope parameter has been set up as an expression to facilitate the scheme we introduced previously to deal with rests. Other simple effects can easily be attained with octave SOUND statements if we de-tune the octaves:

```
70    SOUND 1, envelope, note, tempo*duration
75    SOUND 2, envelope, note + 50, tempo*duration
76    SOUND 3, envelope, note + 97, tempo*duration
```

Using a common percussive envelope, this will give the sound of a West Indian steel band - oil drums tuned to octaves but exhibiting less than perfect tuning.

Exercises

1 Develop two contrasting envelopes (one with a fast percussive attack and one with a slow build-up). Use a

FOR loop to play a tune from a DATA statement twice. The
first time the program should use the first ENVELOPE. The
second time the tune should be played an octave higher
using the second ENVELOPE.

10.8 The ENVELOPE statement: pitch

The pitch envelope permits controlled variation of the
frequency of the sound during its duration. Up to now the
parameters in the pitch envelope have been set to zero. If
these parameters are given non-zero values, the frequency of
the sound initially starts at the specified frequency (given
by the pitch number) and then varies under control of the
pitch envelope parameters. The sound waveform compresses
(frequency increased) and expands (frequency decreases) in
time and this is a form of frequency modulation. The
frequency of the sound waveform is controlled by the
amplitude of another waveform - the pitch envelope.

In music an instrumentalist may cause a slight pitch
fluctuation around a particular note. The way in which this
is achieved varies from instrument to instrument, but the
technique is common with wind and string instruments. It is
called vibrato and the pitch fluctuations are very slight.
Much wilder pitch fluctuations are possible with the pitch
envelope and it can be used to produce 'non-musical'
effects.

The pitch envelope parameters control frequency
variations over three sections and are as follows:

Parameter	Range	Effect
pi1	-128-127	change of pitch number per time step for Section 1
pi2	-128-127	ditto for Section 2
pi3	-128-127	ditto for Section 3
pn1	0-255	number of steps in Section 1
pn2	0-255	number of steps in Section 2
pn3	0-255	number of steps in Section 3

The effect of the pitch envelope lasts for t*(pn1+pn2+pn3)
centiseconds and the envelope normally auto-repeats if its
duration is less than the duration parameter in the
associated SOUND statement. The auto-repeat can be
suppressed by setting the most significant bit of t to 1.
Thus for a time step of 1 and auto-repeat suppressed t=129.
A pair of parameters controls each section. For section 1,
'pi1' pairs with 'pn1', etc.

The following diagram illustrates the significance of the pitch envelope parameters.
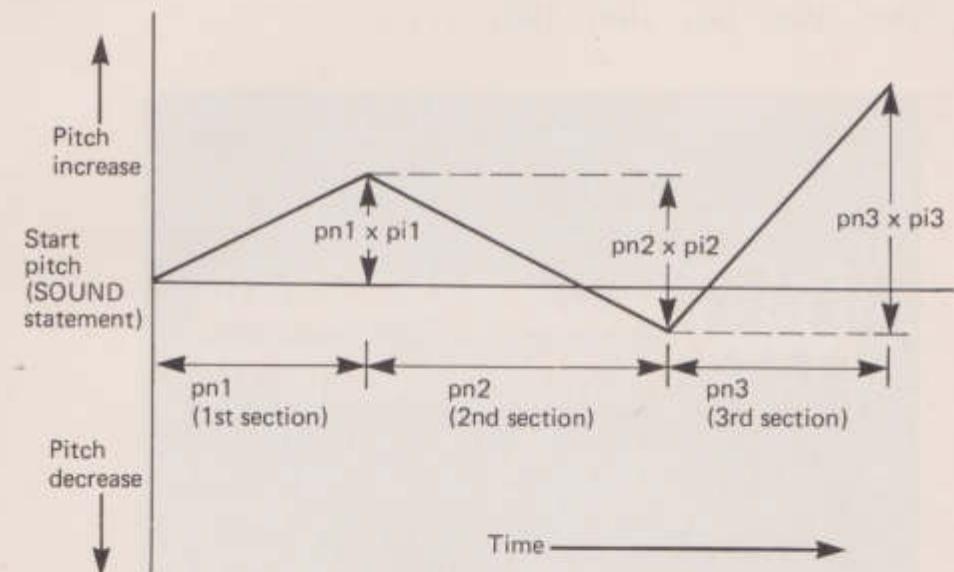
General piecewise-linear form
of a pitch envelope (t = 1). The straight
lines are incremental steps
For example
$$pi1 = 10$$
$$pn1 = 4$$
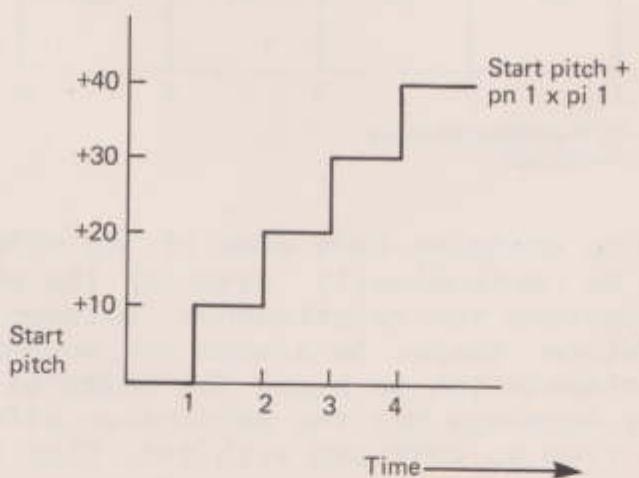
The next illustration is an oscilloscope photograph of a note under pitch envelope control. The parameters used were:

50, -50, 50, 1, 2, 1

and the starting pitch number 100. This results in pitches

100, 150, 100, 50, 100, 150, ...
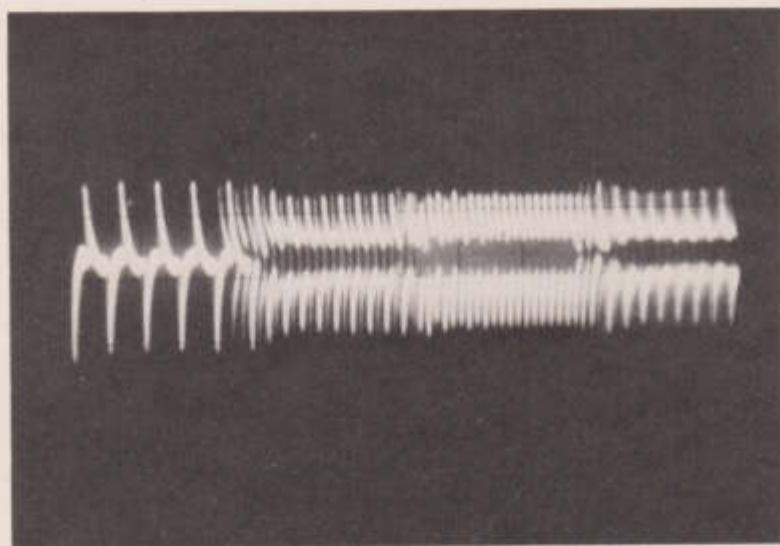
Oscilloscope photograph of a sound under pitch
ENVELOPE control. Pitch ENVELOPE parameters:
50, -50, 50, 1, 2, 1
generating the pitch sequence
100, 150, 100, 50, 100. ...

50	100	150	100
1	2	3	4	5
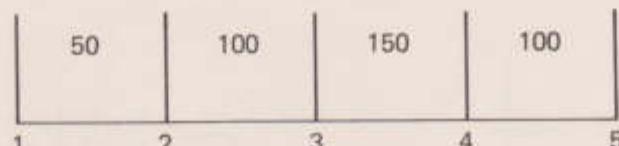
Pitch variation in above
photograph

The following examples show some of the effects that can be achieved by methodically altering the pitch envelope. They also illustrate the relationship between the numbers and the envelope shape. As always for non-musical effects personal experimentation is best. The examples given can be tried with any envelope but the subjective effects described were obtained from an ENVELOPE with t=1. They are best tried out in conjunction with a tune from one of the earlier programs.

	1	2	3
pi	1	-1	1
pn	1	2	1
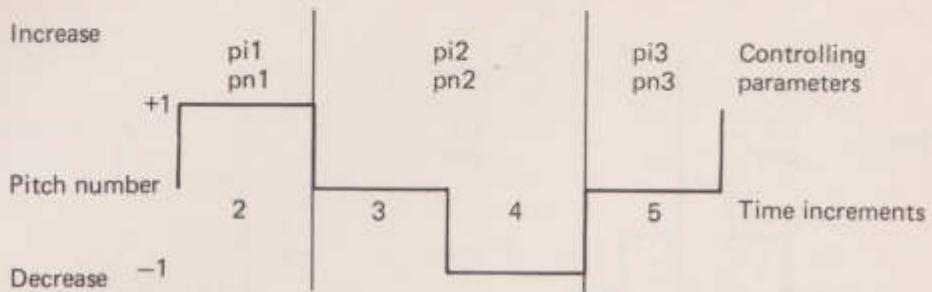
This gives a 'shimmering' effect. The pitch changes are minimal and there is a symmetrical sharpening and flattening of tone.

	1	2	3
pi	8	-8	8
pn	1	2	1
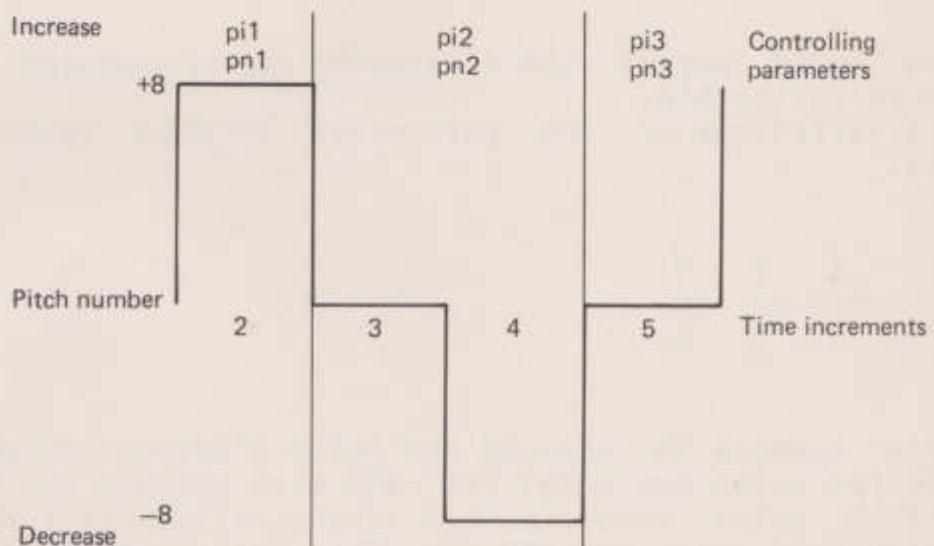
This gives a 'burbling' or 'rasping' effect. There is now excessive flattening and sharpening of the note, but this is not noticeable because it takes place very quickly compared with the 'electronic burbling'.

```
        1    2    3
pi      1   -1    1
pn      4    8    4
```
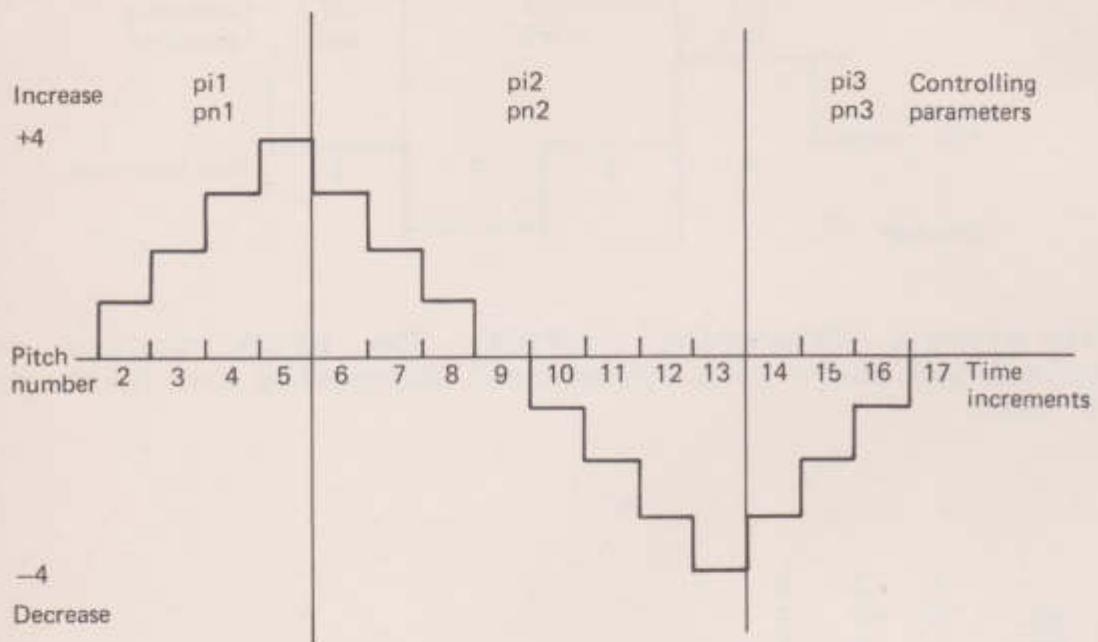

As you would expect the flattening and sharpening is now much more noticeable.

Wild variations of the parameters produce interesting effects.

```
        1    2    3
pi     50  -50   50
pn      2    4    2
```

In this example the effects are quite difficult to predict because the pitch decrement (4x -50) will attempt to reduce the pitch below zero. In this case positive pitch numbers are calculated using MOD 256 (see User Guide). If the effect of the pitch envelope is not to mask the tune, then the pn parameters must be short.

10.9 Special effects

Most special effects are obtained through using channel 0, and most of the material in this section uses this facility. An incidental and useful effect, however, is the use of flushing to produce a glissando, or series of consecutive notes sounded very quickly. The most (ab)used musical

glissando is produced by running a finger over the strings
of a harp. In the next program, which produces a 'arcade
game fanfare', the duration of each note is cut down from
the minimum of 1/20 second to however fast the computer can
flush its previous note and change the current one. The
speed of the glissando can be controlled by inserting a
delay.

```
10    start = 0
20    FOR glissando = 1 TO 20
30      FOR note = start TO start+100
40        SOUND &11, -15, note, 2
50      NEXT note
60      start = start + 4
70    NEXT glissando
```

Now as we mentioned above, channel 0 generates noise;
either periodic (bursts of noise) or white noise (noise
containing many different frequencies - sounds like a
waterfall). The particular noise selected is controlled by
the value of the pitch number as follows.

SOUND   0, loudness, p, duration

p	noise selected
0	High-frequency periodic
1	Medium-frequency periodic
2	Low-frequency periodic
3	Frequency of noise determined by pitch number of channel 1
4	High-frequency white noise
5	Medium-frequency white noise
6	Low-frequency white noise
7	Frequency of (white) noise determined by pitch number of channel 1

Again experimentation with these effects is best left to
you. Here we reproduce two examples. The first is the noise
of machine gun-fire, the second the sound of a spaceship
taking off. The second uses proportional noise and the
channel 1 SOUND statement that controls frequency is silent.

```
10    FOR bullet = 1 TO 10
20      SOUND 0, -15, 3, 1
30      SOUND 0, -15, 4, 1
40    NEXT bullet
```

```
10    FOR i = 1 TO 255
20      SOUND 1, 0, i, 1
30      SOUND 0, -15, 7, 1
40    NEXT i
50    SOUND 0, -15, 7, 200
```

Rhythmic effects

Finally the next program demonstrates the use of channel 0
in conjunction with a percussive envelope to produce a
rhythm from a monotone percussion instrument. The rhythm is:

and if you don't know what this is from reading the dots
then play the program.

```
10    duration = 2
20    ENVELOPE 1,1, 0,0,0,0,0,0,
                 126, -20, -20, -20, 126, 60
30    ENVELOPE 2,1, 0,0,0,0,0,0,
                 90, -20, -20, -20, 90, 60
35    ENVELOPE 3,1, 0,0,0,0,0,0,
                 20, -20, -20, -20, 126,90

40    FOR bar = 1 TO 16
45      FOR triplet = 1 TO 2
50        FOR note = 1 TO 3
60          IF note=1 THEN envelope = 1 ELSE envelope = 2
70          SOUND 0, envelope, 4, duration
80        NEXT note
90      NEXT triplet
100     FOR note = 1 TO 4
110       SOUND 0, 3, 4, duration*3/2
120     NEXT note
150   NEXT bar
```

Note also in this program the use of two ENVELOPE statements
to simulate the fact that the emphasis can be caused either
by an amplitude change or by the note being played on
another drum.

Chapter 11 **Animation**

When film cartoons are made by hand, animated effects are created by drawing and photographing a large number of 'frames' which are then displayed by a projector at a speed that gives the impression of continuous movement.

Computer animation packages now exist that help the cartoon artist to create animated films. Such a package typically includes programs that help the artist to design scenes from the film, using commands for interactively drawing lines and colouring regions. An animation package also includes programs for carrying out tasks such as 'in-betweening', a tedious and time-consuming job carried out by the 'in-betweeners' or junior artists of the cartoon film industry. Here, the main frames of a film are created on the screen by an artist and the hundreds of in-between frames that bridge the gaps between the main frames are generated by the animation package, each frame being photographed as it is created.

If you have the facilities for making films and wish to use your BBC micro for creating cartoons, then the full power of the graphics facilities can be used in drawing each frame of the film. The time taken to change the image on the screen is not critical as each frame of the film will be photographed only when the changes on the screen are complete. It can take many hours of program runs to create a few seconds of film in this way.

The type of animation that is most likely to be of interest to the owner of the BBC micro is the creation of 'real-time' animation effects on the screen. These might be required as an essential element of a game or they might be created simply for entertainment. In such uses of animation, the speed at which the image on the screen changes is critical and this limits the range of programming techniques that can be used effectively. In this chapter, we shall be concentrating on real-time animation.

11.1 A simple moving object

It is important that the programmer who wishes to use real-time animation understands the limitations of the various techniques that can be used. There are two main methods of displaying information on the screen – we can use the various graphics plotting facilities or we can print

characters. Surprisingly, it is the latter facility that is more useful in creating effective animation. In this section, we shall look at the use of these two methods in programs that move a simple object across the screen. The object that we shall use is a very simple 'box-car' shape:
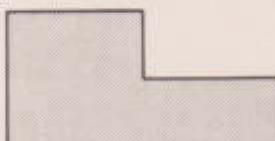
We shall make this shape 'drive' across the screen from left to right.

Using PLOT facilities for animation

The simplest idea that can be used to create animation effects is repeatedly to delete an object and redraw it in a different position. If this happens sufficiently quickly, then the object appears to move. A program to do this has the following outline structure:

```
Set x and y to initial position of object
Draw object at x,y

REPEAT
    Calculate newx, newy
    Delete object at x,y
    Draw object at newx,newy
    x = newx : y=newy
UNTIL final position reached
```

A good general principle that we have kept in mind while writing the above outline is that redrawing should occur as soon as possible after deletion.

One way of deleting an object is to redraw it in the current background colour and this is the method that is used in the rather unsatisfactory program below that implements the above approach.

The parameter of PROCcar that specifies the logical operation to be used in plotting is needed in the next section. Because speed of program execution is often critical in animation, we shall use integer variables (with '%' signs) throughout this chapter (see Appendix 8). It is important that you run this program for different values of 'xstep%' and observe the problems that it illustrates. The most obvious drawback is the flashing effect that is a result of the time taken to delete and redraw even this simple object. However, this defect can be eliminated by using the techniques of the next section.

```
10      INPUT xstep%
20      MODE 5
30      cx% = 0 : cy% = 500 :REM initial car coordinates
40      PROCcar(cx%, cy%, 0, 3)

50      REPEAT
60        nx% = cx% + xstep% :REM new x coordinate
70        PROCcar(cx%,cy%,0,0)
80        GCOL 0,3
90        PROCcar(nx%,cy%,0,3)
                      :REM y coordinate has not changed
100       cx% = nx%
110     UNTIL cx% > 1279

120     MODE 7 : END

310     DEF PROCcar(x%, y%, logop%, colour%)
320       GCOL logop%, colour%
330       MOVE x%,y%
335         REM all plots relative to previous point
340       PLOT 0, 64,0
350       PLOT 81, -64,64
360       PLOT 0, 64,0
370       PLOT 81, 0,-64
380       PLOT 0, 64,0
390       PLOT 81, -64,32
400       PLOT 0, 64,0
410       PLOT 81, 0,-32
420     ENDPROC
```

Frame switching

The flashing effect exhibited by the above program was due
to the fact that we could see the car being erased and
redrawn. In order to eliminate this flashing effect, we need
to arrange for the erasing and redrawing process to take
place without it being seen. To do this, we need to work
with two separate 'frames' and display one frame on the
screen while the erasing and redrawing process is being
carried out in the other frame.

In MODE 5, the colour of each pixel is coded as a two-bit
number. We saw in Chapter 9 that, instead of treating the
screen as a single image plane in which each pixel is one of
four colours, we can treat it as two separate image planes,
or frames, in which each pixel is one of two colours. In
each MODE 5 pixel, one of the two bits is taken to represent
the colour of a pixel in one frame and the other bit is
taken to represent the colour of the corresponding pixel in
the other frame. The alternative significance of each two-
bit colour code is given by the following table:

Single image plane		Two separate image planes	
colour code	bit pattern	frame 1	frame 2
0	00	0	0
1	01	1	0
2	10	0	1
3	11	1	1

To switch between frames 1 and 2, we use VDU 19 statements to associate different combinations of actual colours with our four colour codes. For example, if we want 0 to be the background colour code and 1 to be the foreground colour code in each of frames 1 and 2, then we can selectively display one of the two frames by selecting one of the two actual colour combinations given in the following table:

Colour code	Actual colour settings	
	frame 1 displayed frame 2 hidden	frame 2 displayed frame 1 hidden
0	background	background
1	foreground	background
2	background	foreground
3	foreground	foreground

If we are using the same background and foreground colours in frame 1 as in frame 2, colour code 0 is always set to the background colour and colour code 3 is always set to the foreground. If the background colour is black and the foreground colour is white, then the colour codes 0 and 3 are correctly initialized in mode 5. To switch frame 1 on and frame 2 off, we need only use:

```
VDU 19, 1,7, 0,0,0
VDU 19, 2,0, 0,0,0
```

and to switch frame 1 off and frame 2 on, we use

```
VDU 19, 1,0, 0,0,0
VDU 19, 2,7, 0,0,0
```

A new shape can be plotted in frame 1 by preceding the plotting instructions by

```
GCOL 1,1
```

A shape can be erased from frame 1 by replotting it after

```
GCOL 2,2
```

(see Chapter 9). Similarly, a shape can plotted in frame 2 by preceding the plotting instructions by

```
GCOL 1,2
```

and erased by redrawing the shape after

```
GCOL 2,1
```

The following is an outline of how we use the above technique to conceal the deleting and redrawing process while an object is being moved about the screen:

```
Set x,y to the initial position of the object
Switch frame 1 on, frame 2 off
Draw shape in frame 1

REPEAT
    Calculate newx,newy
    Draw shape at newx,newy in off frame
    Switch frames
    Erase car at position x,y in frame that is now off
    x=newx : y=newy
UNTIL final position reached
```

In order to drive our box car across the screen until it is out of sight, the details are filled in as follows:

```
10      INPUT xstep%
20      MODE 5
30      cx%=0 : cy%=500
40      PROCswitchon(1)
50      PROCcar(cx%, cy%, 1, on%)

60      REPEAT
70        newcx% = cx%+xstep%
80        PROCcar(newcx%, cy%, 1, off%)
90        PROCswitchon(off%)
100       PROCcar(cx%, cy%, 2, on%)
105                 REM deletes car in off frame
110       cx% = newcx%
120     UNTIL cx% > 1279

130     MODE 7 : END

210     DEF PROCswitchon(frame%)
220       on% = frame% : off% = 3-on%
230       VDU 19, on%, 7, 0,0,0
240       VDU 19, off%,0, 0,0,0
250     ENDPROC

310     DEF PROCcar(x%, y%, logop%, colour%)
          ... as before
```

Now that we have eliminated the flashing effect, we turn our attention to the speed at which the car moves. The speed of motion can be varied by increasing the steps in which it moves. With xstep=64, the car takes just over 2 seconds to move across the screen. Taking bigger steps makes the motion rather 'jumpy'. This may seem a reasonable speed, but you should bear in mind that our program is doing nothing else but moving the car. If we were to use PLOT facilities to implement the animation for a game in which several objects such as spaceships were being moved about the screen and in which other calculations were taking place, then the rate of motion for each of the objects would deteriorate to an unacceptable level. In general, line plotting and colour filling facilities are too slow for most real-time animation. These facilities can, however, be used to good effect for creating backgrounds for character animation.

Using character printing to animate an object
In any of the graphics modes, a character shape can be displayed by a PRINT statement in considerably less time than it would take to draw the same shape using graphics commands. This is because special hardware is used to fill the area of the display buffer that is to be occupied by the character. We shall see later in the chapter how to define our own character shapes. In this section, we introduce the idea of using character printing to create moving objects. First, let us make a single character move horizontally across the screen as quickly as possible. The following program represents one way of doing this:

```
10    MODE 5
20    PRINT TAB(0,10); "*";
30    TIME = 0

40    FOR x%= 0 TO 18
50      PRINT TAB(x%,10); " *";
60    NEXT x%

70    PRINT TIME
```

Note that we have combined the process of deleting and reprinting the character in a single PRINT statement. The string printed consists of a space followed by a star. The space obliterates the old star at almost the same instant that the new one is printed. Care must be taken that printing does not take place beyond the end of the line or the star will appear at the beginning of the next line.

If you run this program, you will find that it takes only a few hundredths of a second for the star to cross the screen. If you want to see it, you will have to insert a delay loop between lines 40 and 50.

In applications where speed is a problem, for example if we are moving a large number of objects, there are often tricks that can be used to increase program speed. The above program can be speeded up slightly by using the backspace character instead of TAB as follows:

```
10    MODE 5
20    s$ = CHR$(8)+" *"    :REM 8 is code for backspace.
30    PRINT TAB(0,10); "*";
40    TIME = 0

50    FOR x% = 0 TO 18
60      PRINT s$;
70    NEXT x%

80    PRINT TIME
```

The slight improvement in speed that this represents might be important in a complex program. As a rough rule of thumb, a TAB operation takes the same time as it does to print three or four backspaces. Note that in this case the semicolons at the end of lines 30 and 60 are essential.

Now let us return to our simple box-car shape. We can build this up in MODE 5 from characters, each of which is a solid block of colour, as follows:
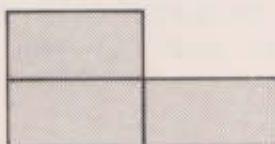
If necessary, we could define such a character using the VDU 23 command described in a later section, but you will find that, outside MODE 7, some of the so-called 'undefined' character codes are in fact preset as solid blocks of colour. For example, in our program we have used the character whose code is 240. (At least this works on our machine - you may have to use a different code.) Here is a program to move the car across the screen:

```
10    MODE 5
20    blob$ = CHR$(240)
30    cara$ = blob$ : carb$ = blob$+blob$
40    PRINT TAB(0,10); cara$; TAB(0,11); carb$;
50    ca$ = " "+cara$ : cb$ = " "+carb$

60    FOR x%=0 TO 17
70      PRINT TAB(x%,10); ca$; TAB(x%,11); cb$;
80    NEXT x%
```

Although there are two characters in the bottom half of the car, only one leading space is needed in the string that deletes and reprints this part of the car - the image is being moved only one character position at a time. Because of the speed at which characters are printed, there is no appreciable flashing and the car moves across the screen in about one fifth of a second.

You will find the presence of the flashing text cursor annoying - it hovers round the object that is being moved. In the first version of the operating system, it can be switched off by adding the following line to the program:

```
11    VDU 23; 8202; 0;0;0;
```

In later versions of the operating system, you can use

```
VDU 23,1, 0;0;0;0;   to turn off the cursor
VDU 23,1, 1;0;0;0;   to turn on  the cursor
```

11.2 Diagonal motion - a bouncing ball

In the last section, the objects that we animated moved only horizontally. We can create vertical motion by changing only the y-coordinate of the object and diagonal motion can be obtained by changing both x and y coordinates together. To illustrate this we shall write a program that creates the effect of a ball moving diagonally on the screen and bouncing at the edges. The program will terminate if someone hits the space bar. The main part of the program will be:

```
10    MODE 4
20    PROCinitialise
30    PROCnewball

40    REPEAT
50      PROCmoveball
60      k$ = INKEY$(0)
70    UNTIL k$ = " "

80    MODE 7 : END
```

The INKEY$ function can be used if we want a program to look and see if a key has been pressed. The parameter in brackets indicates how long the program should wait if a key has not been pressed. The parameter 0 indicates that there should be no delay. If no key has been pressed, then the result of the function is the empty string, otherwise it is the character that has been typed. Thus our program will terminate and revert to MODE 7 if we hit the space bar.

PROCinitialise and PROCnewball are defined as follows:

```
100     DEF PROCinitialise
110       VDU 23; 8202; 0;0;0;
120       ball$ = "o"
200     ENDPROC

210     DEF PROCnewball
220       bx%=RND(38) : by%=1 :REM ball coordinates
230       PRINT TAB(bx%,by%); ball$;
240       xdir%=1 : ydir%=1 :REM initial directions
300     ENDPROC
```

We have separated the initialisation process into two separate procedures for reasons concerned with later extensions of the program. The ball starts in a random position at the top of the screen. The variables 'xdir%' and 'ydir%', which are initialised to 1, represent the values that are to be added to 'bx%' and 'by%' each time the ball is moved. Thus the ball will start moving diagonally downwards towards the right.

Whenever the ball reaches one of the sides of the screen, the value of 'xdir%' must be changed and whenever it reaches the top or bottom of the screen, the value of 'ydir%' must be changed. For example, if xdir%=1, ydir%=1 and the ball reaches line 28 of the screen, then the statement:

ydir% = -1

should be obeyed so that the ball will subsequently be moved by adding 1 to its x-position and adding -1 to its y-position (until another direction change takes place).


```
                xdir%=1                      xdir%=1
                  ydir%=1                    ydir%=-1




        screen line 28 — — — — — — — — — — —
```
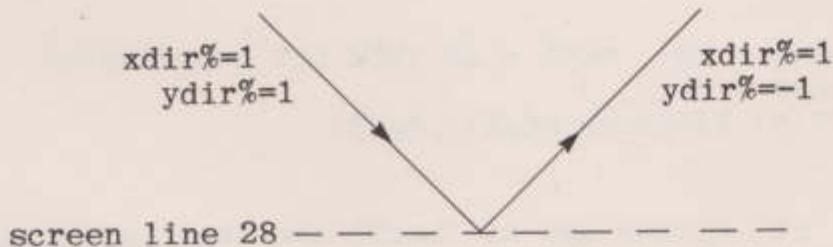
PROCmoveball can be defined as follows:

```
310     DEF PROCmoveball
320       LOCAL nx%,ny% :REM new ball coordinates
330       IF bx%=1 THEN xdir%=1 ELSE IF bx%=38 THEN xdir%=-1
340       IF by%=1 THEN ydir%=1 ELSE IF by%=28 THEN ydir%=-1
350       nx% = bx% + xdir% : ny% = by% + ydir%
360       PRINT TAB(bx%,by%); " "; TAB(nx%,ny%); ball$;
370       bx% = nx% : by% = ny%
400     ENDPROC
```

We have left an unused margin round the edge of the screen for use in later extensions of the program. When doing character animation, it is advisable to leave an unused margin of at least one line along the bottom of the screen. This is because we must avoid, at all costs, the possibility of printing a character in the position at the bottom right-hand corner of the screen. Even if the PRINT statement were terminated by a semicolon, the computer would prepare to print the next character at the beginning of the next line and would 'scroll' up the whole screen to make room for it.

If you run the above program, you will find that the ball moves round the screen very quickly. There are two obvious ways in which it could be slowed down. We could insert a delay loop between lines 40 and 50, or we could simply increase the parameter of INKEY$ at line 60. We shall find that both of these methods are unsatisfactory when we introduce a bat into the program in the next section. A more satisfactory method of speed control will be discussed there.

11.3 Controlling movement from the keyboard: a bat'n'ball program

An essential requirement in video games is that the person playing the game should be able to control the movement of guns, bats, spaceships or 'munchies' from the keyboard. Let us illustrate this by extending the bouncing ball program so that a 'bat' moves in either direction across the bottom of the screen under the control of someone at the keyboard. We can extend PROCinitialise to position the bat centrally by including the following instructions.

```
130     btx% = 16 : bty% = 29 :REM bat coordinates
140     bat$ = " ____ "
150     PRINT TAB(btx%,bty%); bat$;
```

The spaces at either end of 'bat$' are used to delete any part of the old bat that is left behind whenever it is moved one step to the left or to the right.

In order to increase the sensitivity of the keys that will control the movement of the bat, we insert commands in PROCinitialise to change the auto-repeat timings for the keys:

```
160     *FX 11,5
170     *FX 12,5
```

These commands were used in Chapter 10 and their effect is summarised in Appendix 10.

We prepare to recognise that the ball has gone out of 'court' by introducing a logical variable 'missed%' and initialising it in PROCnewball:

```
250      missed% = FALSE
```

The main part of the program is now amended to:

```
10    MODE 4
20    PROCinitialise
30    PROCnewball

40    REPEAT
50      PROCmovebat
60      PROCmoveball
70    UNTIL missed%

80    *FX 12,0
90    MODE 7 : END
```

The *FX command at line 80 resets the auto-repeat timings to their normal values.
PROCmovebat is defined as:

```
410    DEF PROCmovebat
420    LOCAL k$ :REM bat control key
430      k$ = INKEY$(0)
440      IF k$ = "Z" THEN btx% = btx%-1
           ELSE IF k$ = "X" THEN btx% = btx% + 1
           ELSE ENDPROC
450      IF btx%<0 THEN btx% = 0
           ELSE IF btx%>35 THEN btx% = 35
460      PRINT TAB(btx%,bty%); bat$;
500    ENDPROC
```

The keys used to control the motion of the bat are a matter of personal choice. Previously, when the ball reached the bottom of the screen, it bounced, but now we want it to bounce only if the bat is in position to hit it. Once PROCmoveball has calculated the new ball coordinates, it must check whether the ball is on the same row as the bat, and, if it is, it must check whether the bat is in position to hit the ball. We need:

```
340      IF by%=1 THEN ydir%=1
350      nx% = bx% + xdir% : ny% = by% + ydir%
355      IF ny% = bty% THEN PROCtryhit
```

where PROCtryhit is defined as:

```
510    DEF PROCtryhit
520      IF btx%<nx% AND btx% + 4 > nx%
            THEN ny% = bty%-1 : ydir% = -1 :
                           SOUND 1,-15,101,5
530    ENDPROC
```

IF the bat misses the ball, we leave 'ydir%' set to +1 and allow the ball to move on past the bat. We need to trap the ball before it goes off the bottom of the screen and this can be done by inserting the following line at the start of PROCmoveball:

```
328      IF by%=30 THEN
            PRINT TAB(bx%,by%);" "; : missed%=TRUE : ENDPROC
```

Now we consider the problem of slowing the ball down to a reasonable speed. The two obvious solutions mentioned earlier are rather unsatisfactory for different reasons.

If we change line 430 to:

```
430      k$ = INKEY$(10)
```

then as long as the bat is not being moved, there will be a delay of 10 hundredths of a second between each move of the ball. However, if we hold down one of the bat control keys in order to move the bat quickly, then the calls of INKEY$ will not have wait for values. This will have the effect of reducing the delay and speeding up the ball.

The other obvious way of slowing down the ball is to insert a delay loop somewhere in the main loop between lines 40 and 70. This would keep the speed of the ball constant, but it would slow down the program's response to the bat control keys which would be undesirable.

You should try both of the above methods for yourself and observe the effects described.

Timing animation events

We now describe a method for controlling the speed of one object in an animation program without creating unwanted side-effects on the motion of other objects in the same program. Each time an object is moved, the time at which the next move should take place is calculated and the object should not be moved again until TIME has passed this calculated value. For example, if we want a time delay of 10 hundredths of a second between each move of the ball, we introduce a variable 'btime%' whose value indicates the time at which the next move is to take place. We extend PROCinitialise to include:

```
125      bdelay% = 10 : btime% = bdelay%
126      TIME = 0
```

PROC moveball should now do nothing and exit immediately if it is not yet time to move the ball. When the ball is moved, 'btime%' should be reset to a new value. We require to insert the following instructions at the start of the procedure:

```
322      IF TIME<btime% THEN ENDPROC
324      btime% = TIME + bdelay%
```

We are ensuring that the program will not activate an animation event until an appropriate time interval has elapsed since the last activation of the same event. Note that we can never guarantee that the time interval between events is exact. When this technique is being used, the variable TIME must not be altered by other parts of the program.

Exercises

1   Although the PLOT graphics statements are usually too slow for animation, they are very useful for drawing backgrounds. We have left a one-character margin round the top and sides of the 'court' in the bat'n'ball program. Use graphics commands to draw sides and a back for the court before the game starts.

2   Decide on a scoring system for the bat'n'ball program. For example, points could be scored each time the bat hits the ball, or the points scored could depend on the time that the ball is kept in play. An up-to-date score can be displayed and continuously reprinted on line 31 of the screen which has been kept clear for this purpose. NOTE: Any PRINT statement that displays information on line 31 <u>must</u> terminate with a semicolon, otherwise the whole court will scroll up each time the score is printed. As mentioned earlier, you must also avoid printing anything in the last character position on this line.

3   The section of our bat'n'ball program between lines 30 and 70 plays a game with one ball until it goes out of court These lines could themselves be repeated:

```
25   REPEAT
        :
        :
75   ballsleft% = ballsleft%-1
76   UNTIL ballsleft%=0
```

with appropriate initialisation of 'ballsleft%'. Make

this modification to the program and extend the display on line 31 of the screen to include an up-to-date record of the number of balls left.

4 Allow the user to select different levels of play (with different speeds of ball) and set 'bdelay%' accordingly.

5 Add some further sound effects to the program.

11.4 A 'space-attacker' program

In this section, we present the outline of a very simple 'space-attacker' program. This program will demonstrate a combination of three different sorts of movement:

(a) A gun, controlled in exactly the same way as the bat in the previous section, will move backwards and forwards across the bottom of the screen.

(b) Characters representing enemy 'spaceships' will repeatedly appear at the top of the screen and will attempt to get past the gun at the bottom without being shot down. They will continuously take evading action and each spaceship will move faster than the previous one.

(c) When the gun is fired (by hitting the space bar), a bullet track or laser effect will be created by very rapid printing and erasing of a column of characters.

For the moment, an attacker will be represented by "*" and the gunsight by the letter "A". We shall see in the next section how to define more appropriate shapes for these characters.

The main part of the program is:

```
10    MODE 4
20    PROCinitialise

30    REPEAT
40       PROCmakeattack
50    UNTIL lives%=0

60    *FX 12,0
70    MODE 7
80    PRINT TAB(10,10); "GAME OVER"
90    PRINT TAB(10,12); "SCORE: "; points%
100   END
```

PROCinitialise is defined as:

```
110    DEF PROCinitialise
120      *FX 11,5
130      *FX 12,5
140      VDU 23; 8202; 0;0;0;
150      gx% = 20 : gy% = 30
160      gun$ = "A"
170      PRINT TAB(gx%,gy%); gun$;
180      adelay% = 100 : TIME = 0
190      points% = 0 : lives% = 10
200      att$ = "*"
210      b$ = CHR$(11)+"¦"+CHR$(8):REM for bullet track
220      e$ = CHR$(11)+" "+CHR$(8):REM for track erase
230    ENDPROC
```

The use of 'b$' and 'e$' in creating the effect of a bullet track will be discussed shortly.

PROCmakeattack consists of a loop that is very similar to the main loop in our bat'n'ball program:

```
300    DEF PROCmakeattack
310      PROCnewattacker
320      REPEAT
330        PROCattacker
340        PROCgun
350      UNTIL attackover%
360    ENDPROC
```

Each time a new attacker is created, the speed will be increased. We do this by reducing the value of 'adelay%' by one tenth. Each new attacker starts at a random position on the top row of the screen:

```
400    DEF PROCnewattacker
410      ax% = RND(40)-1 : ay% = 0
420      PRINT TAB(ax%,ay%); att$;
430      adelay% = adelay% - adelay%/10
440      atime% = TIME + adelay%
450      attackover% = FALSE
460    ENDPROC
```

The procedure PROCattacker will be very similar in style to PROCmoveball in the last section. The main difference is in the way that the new attacker coordinates are calculated from the old ones. The attacker always moves down one step in the y-direction. However, we want it to take avoiding action as the player attempts to aim the gun at it. If the attacker is at the same x-position as the gun (line 560), then the program makes a random choice of whether it should

move left, right or stay in the same x-position. If the
attacker is not on the same vertical line as the gun, then
its x-coordinate is changed so as to move it away from the
gun. The expression 'SGN(ax% - gx%)' has a value of +1 or -1
depending on which side of the gun the attacker is.

```
500     DEF PROCattacker
510     LOCAL nx%, ny% :REM new coordinates for attacker
520       IF TIME < atime% THEN ENDPROC
530       atime% = TIME + adelay%
540       IF ay% = gy% THEN
              lives% = lives%-1 : PRINT TAB(ax%,ay%); " "; :
              attackover% = TRUE : ENDPROC
550       ny% = ay% + 1
560       IF ax% = gx% THEN nx% = ax% + RND(3) - 2
                  ELSE nx% = ax% + SGN(ax% - gx%)
570       IF nx%<0 THEN nx%=0  ELSE  IF nx%>39 THEN nx%=39
580       PRINT TAB(ax%,ay%); " "; TAB(nx%,ny%); att$;
590       ax% = nx% : ay% = ny%
600     ENDPROC
```

The above procedure takes no account of the possibility of a
collision between the attacker and the gun. We leave this
aspect of the program as an exercise.

The procedure PROCgun will respond to the keys for
controlling its movements in the same way as the bat control
procedure of the last section. Since the gun consists only
of a single character, it is convenient if the value of
'gx%' represents the horizontal position of the gun (and not
the position of a preceding space as was the case with the
bat). We also have one further command key to cater for:

```
610     DEF PROCgun
620     LOCAL k$, nx% :REM command key and new gun position
630       k$ = INKEY$(0)
640       IF k$=" " THEN PROCfire : ENDPROC
              ELSE IF k$ = "Z" THEN nx% = gx% - 1
              ELSE IF k$ = "X" THEN nx% = gx% + 1
              ELSE ENDPROC
650       IF nx%<0 THEN nx%=0 ELSE IF nx%>39 THEN nx%=39
660       PRINT TAB(gx%,gy%); " "; TAB(nx%,gy%); gun$;
670       gx% = nx%
680     ENDPROC
```

To create the effect of gunfire, the program will calculate
the range of the shot and print a column of "¦" characters
from the gun either to the attacking spaceship or to the top
of the screen. It will then erase these characters by
printing spaces in exactly the same way. Because of the

speed at which this happens, there is no appreciable slowing down of the rest of the program.

```
710     DEF PROCfire
720     LOCAL r% :REM range of shot
730       IF gx%=ax% THEN r% = ay% : points% = points%+10 :
                            attackover% = TRUE
                      ELSE r% = 0
740       PROCtrack(r%, b$)
750       PROCtrack(r%, e$)
760     ENDPROC
800
810     DEF PROCtrack(r%, s$)
820     LOCAL y%
830       PRINT TAB(gx%, gy%);
840       FOR y% = 29 TO r% STEP - 1
850         PRINT s$;
860       NEXT y%
870     ENDPROC
```

Each character in our bullet track is printed by using a sequence of three characters. In the string b$, the character to be printed is preceded by CHR$(11) which moves the print position up one line and is followed by CHR$(8) which backspaces to the position of the character just printed. Starting off at the position of the character representing the gun means that the first character of the bullet track is printed immediately above the gun, the next immediately above that and so on. Carrying out the same process with string 'e$' overprints the bullet track with spaces. Of course the semicolon at the end of the PRINT statement on line 850 is vital.

Exercises

1 Introduce a more complicated scoring scheme into the attacker program. For example, the points scored could vary with distance from the gun. Display an up-to-date record of the progress of the game on line 31 of the display - points so far and number of attackers that have penetrated the defences.

2 No account is taken in our program of the possibility of the attacker colliding with the defender's gun. If this happens it should terminate the game. Modify PROCattacker so that it deals with such an occurrence.

3 Add appropriate sound effects to the program.

11.5 User-defined characters

In modes 0, 1, 2, 4 and 5, the screen is divided up into a number of 'pixels' (see Chapter 9). For example, in MODE 4, there are 320x256 pixels.

In modes 3 and 6, the screen is divided into horizontal strips of pixels which are separated by strips of background colour. Each strip is 8 pixels deep.

In any of modes 0 to 6, printing a character has the effect of filling an 8x8 group of pixels with a pattern of foreground and background colour. For example, the pattern for "A" is:
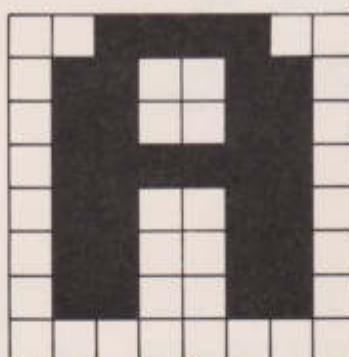
Also associated with each character is an ASCII code number in the range 0 to 255. This code is used inside the computer to represent the character. The ASCII code for "A" is 65. When the character whose code number is 65 is to be displayed on the screen by a PRINT statement, the above pattern of foreground and background colour is inserted into the 'refresh buffer' that contains information about what is currently displayed on the screen.

The user is free to define the character shapes that are associated with ASCII code numbers 224 to 255, and this is particularly useful when creating shapes for use in animation. In fact, on later versions of the machine operating system, it is possible for the user to define shapes for a much greater range of ASCII codes.

Once a new character shape has been defined, it can be displayed on the screen at the same speed as the predefined characters that we have used so far in this chapter.

The use of user-defined character shapes has two advantages over the use of PLOT instructions to draw shapes. Firstly, as we have already seen, a character shape is displayed on the screen at a much greater speed than can be achieved by using PLOT facilities. Secondly, the sequence of PLOT statements needed to draw a complex shape such as a spaceship would be rather lengthy.

Single character shapes

Let us demonstrate the process of defining a new character shape by defining a spaceship for use in the space-attacker program of the last section. The shape we shall define appears below together with a photograph showing several copies of the spaceship in MODE 5. Note that in MODE 5, a pixel (and therefore a character) is elongated.
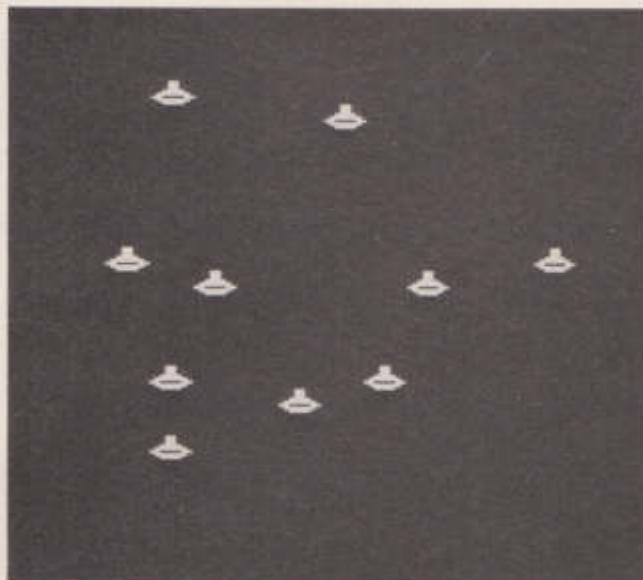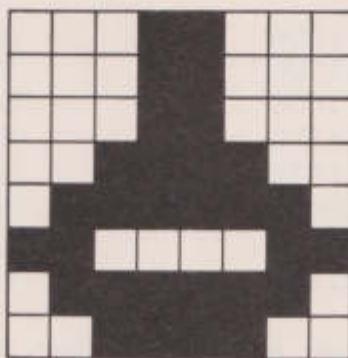
Each row in the 8x8 pattern can be viewed as a byte (eight bits – see Appendix 5), and each byte can be written as an integer in the range 0 to 255. Thus the above pattern can be described as a list of 8 bytes or a list of 8 integers:

bytes	integers
00011000	24
00011000	24
00011000	24
00111100	60
01111110	126
11000011	195
01111110	126
00111100	60

A byte can also be written in the form of two hexadecimal digits. Four bits correspond to one hexadecimal digit as described in Appendix 5. Because of this correspondence, it is usually easier to write a pattern of 8 bits in hex than to convert it into an integer:

bytes	hex
00011000	&18
00011000	&18
00011000	&18
00111100	&3C
01111110	&7E
11000011	&C3
01111110	&7E
00111100	&3C

In order to define our new character shape, we must choose the ASCII code that we are going to use for the character and we must then calculate the sequence of 8 integers (in decimal or hex) that describes its shape. The ASCII code is associated with the required shape by using the VDU 23 command. For example, if we want ASCII character number 224 to appear on the screen as the above shape, we can use

```
10   VDU 23, 224, &18,&18,&18,&3C,&7E,&C3,&7E,&3C
```

We could equally describe the shape by writing the bytes in decimal:

```
10   VDU 23, 224, 24,24,24,60,126,195,126,60
```

We can display this spaceship in the centre of the screen in MODE 4 by:

```
20   MODE 4
30   PRINT TAB(20,10); CHR$(224)
```

We can use this spaceship in our space attacker program by making two simple changes to PROCinitialise. We must define our character and redefine 'att$':

```
195     VDU 23, 224, &18,&18,&18,&3C,&7E,&C3,&7E,&3C
200     att$ = CHR$(224)
```

We could also define a new shape for the gun:
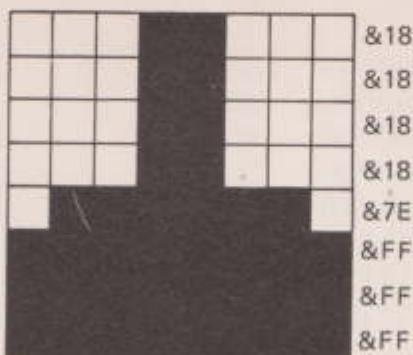
and include the definition of this shape in PROCinitialise
by:

```
155      VDU 23,  225,  &18,&18,&18,&18,&7E,&FF,&FF,&FF
160      gun$ = CHR$(225)
```

Composite character shapes

We can build up bigger objects by defining a number of
different character shapes that can be printed together to
make up the overall shape of the object. For example, we can
slightly improve the shape of the car that was used in the
program of Section 11.1 by defining three separate
characters, one for the bonnet and front wheel, one for the
body and back wheel and one for the roof:
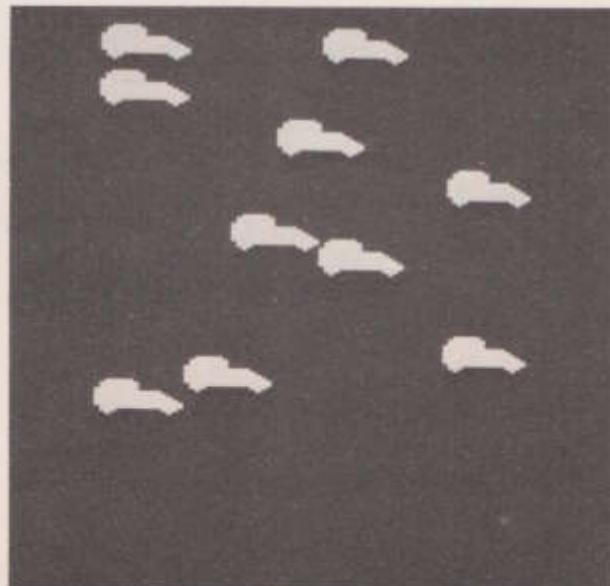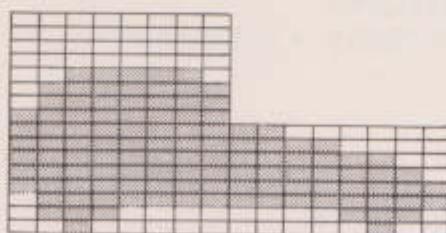
The three characters required are defined by the bit patterns:

```
00000000    &0
00000000    &0
00000000    &0
00000000    &0
00111110    &3E
01111111    &7F
01111111    &7F
11111111    &FF
```

```
11111111    &FF        11000000    &C0
11111111    &FF        11110000    &F0
11111111    &FF        11111100    &FC
11111111    &FF        11111110    &FE
11111111    &FF        11111111    &FF
01111111    &7F        11111111    &FF
01110000    &70        00001110    &E
00100000    &20        00000100    &4
```

We can use these character shapes in the program at the end of Section 11.1 as follows:

```
 5    VDU 23, 224, &0; &0; &3E,&7F, &7F,&FF
 6    VDU 23, 225, &FF,&FF,&FF,&FF,&FF,&7F,&70,&20
 7    VDU 23, 226, &F0,&F8,&FC,&FE,&FF,&FF,&E,4
30    cara$ = CHR$(224)
31    carb$ = CHR$(225) + CHR$(226)
```

A character design program

Designing characters on paper is a tedious process and it is sometimes useful to have a computer program that assists us in the design process. We now present a complete listing of such a program.

The user of this program specifies the size of image he wants to construct (up to 4 characters across and up to 3 characters down). The program then displays a grid on the screen and the user can move a flashing cursor around this grid under the control of the four arrows that are normally used for screen editing. The normal behaviour of these keys is switched off by the line

```
10    *FX 4,1
```

and back on again by the line

```
240    *FX 4,0
```

The current point on the grid is switched to foreground
colour by the key F and to background by the key B. At any
time, the group of characters can be seen at its proper size
in any of the modes 0 to 6 by pressing the corresponding
numeric key. Typing V will display the VDU commands needed
to define the characters. Typing S will save these VDU
commands on a file and they can be later absorbed into a
program using the *EXEC commamd (see Appendix 10). Typing R
will retrieve a complete character image previously saved
using the command S.

The program is terminated by typing "Q" (for Quit). If
this is done in a state where the VDU commands are on the
screen, these will remain on the screen and can be edited
into a program by using cursor editing (as a quick
alternative to *EXEC).

The structure of the program would have been improved if
the various MODE commands had been tidied away inside the
various procedures. However, a MODE statement cannot be
obeyed inside a procedure.

```
10    *FX 4,1
20    MODE 7
30    DIM pic(4,3,8)
40    PROCinitialise

60    REPEAT
70      IF redisplay THEN MODE 7 : PROCdisplay
          ELSE IF planning THEN PROCflash

90      IF c$=up$    THEN PROCup
100     IF c$=down$  THEN PROCdown
110     IF c$=right$ THEN PROCright
120     IF c$=left$  THEN PROCleft
130     IF c$="F" THEN foreground=TRUE
140     IF c$="B" THEN foreground=FALSE
150     IF c$="V" THEN MODE 7 :  PROCvducodes
170     IF c$="S" THEN MODE 7 :  PROCsave
180     IF c$="R" THEN MODE 7 :  PROCrestore
190     IF c$="I" THEN MODE 7 :  PROCinitialise
200     IF c$="H" THEN MODE 7 :  PROChelp
210     IF "0"<=c$ AND c$<="6" THEN
          m=ASC(c$)-48 : MODE m : PROCshowpic
        ELSE m=7
220   UNTIL c$="Q"
230   IF m<7  OR planning THEN MODE 7
240   *FX 4,0
250   END
```

```
270     DEF PROCflash
280       PRINT TAB(col*8-bit+3,(row-1)*8+subrow-1);
290       mask=2^bit
300       IF foreground THEN
              pattern=pic(col,row,subrow) OR mask :
              currch$=on$
          ELSE pattern=pic(col,row,subrow) AND NOT mask :
              currch$=off$
310       pic(col,row,subrow)=pattern
320       REPEAT
330         PRINT on$;back$;
340         TIME=0:REPEAT:UNTIL TIME>10
350         PRINT off$;back$;
360         c$=INKEY$(10)
370       UNTIL c$<>""
380       PRINT currch$; back$;
390     ENDPROC

410     DEF PROCdisplay
420     LOCAL r,sr,c,b
430       CLS
440       FOR r = 1 TO rows
450       FOR sr=1 TO 8
460         PRINT startline$;
470         FOR c=1 TO cols
480         FOR b=7 TO 0 STEP-1
490           IF (2^b AND pic(c,r,sr))=0 THEN PRINT off$;
              ELSE PRINT on$;
500         NEXT:NEXT
510         PRINT endline$
520       NEXT:NEXT
530       PRINT startline$;
540       FOR b=1 TO 8*cols:PRINT"=";:NEXT
550       PRINT endline$;
560       planning=TRUE
570       redisplay=FALSE
580     ENDPROC

600     DEF PROCinitialise
610     LOCAL r,sr,c
620       PRINT:PRINT:PRINT
630       INPUT "Size of pic, cols",cols
640       INPUT "              rows",rows
650       IF cols>4 OR rows>3 THEN
              PRINT "SORRY, TOO BIG!" : END
660       on$=" " : off$=CHR$(255) : back$=CHR$(8)
670       up$=CHR$(139) :   down$=CHR$(138)
680       right$=CHR$(137):left$=CHR$(136)
690       startline$ = CHR$(129)+CHR$(157)+CHR$(135)+""
700       endline$ = ""+" "+CHR$(156)
```

```
710        FOR r=1 TO rows
720          FOR sr=1 TO 8
730            FOR c=1 TO cols
740              pic(c,r,sr)=0
750        NEXT:NEXT:NEXT
760        col=1:bit=7:row=1:subrow=1
770        foreground=FALSE
780        nondispcodes$="0123456VSRIH"
790        PROCdisplay
800      ENDPROC

820      DEF PROCup
830        IF subrow<>1 THEN subrow=subrow-1
           ELSE IF row<>1 THEN row=row-1 : subrow=8
840      ENDPROC

860      DEF PROCdown
870        IF subrow<>8 THEN subrow=subrow+1
           ELSE IF row<>rows THEN row=row+1 : subrow=1
880      ENDPROC

900      DEF PROCright
910        IF bit<>0 THEN bit=bit-1
           ELSE IF col<>cols THEN col=col+1 : bit=7
920      ENDPROC

940      DEF PROCleft
950        IF bit<>7 THEN bit=bit+1
           ELSE IF col<>1 THEN col=col-1 : bit=0
960      ENDPROC

980      DEF PROCshowpic
990      LOCAL r,c,sr,KEY
1000       planning=FALSE : chno=223
1020       FOR r=1 TO rows
1030       FOR c=1 TO cols
1040         chno=chno+1 : VDU 23,chno
1060         FOR sr=1 TO 8
1070           VDU pic(c,r,sr)
1080         NEXT:NEXT:NEXT

1110       PRINT TAB(0,5);
1120       chno=224
1130       FOR r=1 TO rows
1140         PRINT TAB(5);
1150         FOR c=1 TO cols
1160           VDU chno : chno=chno+1
1180         NEXT c : PRINT
1200       NEXT r
1210       PRINT:PRINT:PRINT
1220       c$=GET$
1230       IF INSTR(nondispcodes$,c$)=0 THEN redisplay=TRUE
1240     ENDPROC
```

```
1260    DEF PROCvducodes
1270    LOCAL r,c,sr
1280      planning = FALSE : CLS
1300      FOR r=1 TO rows
1310      VDU 129 : PRINT " Row "; r
1320      FOR c=1 TO cols
1330        PRINT "VDU 23,";
1340        FOR sr=1 TO 8
1350          PRINT ",&";~pic(c,r,sr);
1360        NEXT sr : PRINT
1380      NEXT:NEXT
1390      c$=GET$
1400      IF INSTR(nondispcodes$,c$)=0 THEN redisplay=TRUE
1410    ENDPROC

1430    DEF PROCsave
1440    LOCAL r,c,sr,byte,channel,vdu$,fname$,code,lineno
1450      planning=FALSE
1460      PRINT:PRINT:PRINT
1470      INPUT "Filename",fname$
1480      INPUT "Character code",code
1490      INPUT "Program line number",lineno
1500      channel = OPENOUT(fname$)
1510      FOR r=1 TO rows
1520        FOR c=1 TO cols
1530          vdu$ = STR$(lineno)+" VDU 23,"+STR$(code)
1540          FOR sr = 1 TO 8
1550            byte=pic(c,r,sr)
1560            vdu$=vdu$+",&" + FNhex(byte DIV 16)
                                + FNhex(byte MOD 16)
1570          NEXT sr
1580          PRINT vdu$
1590          PROCoutstring(channel,vdu$)
1600      code=code+1 : lineno=lineno+10
1610      NEXT:NEXT
1620      CLOSE# channel
1630      PRINT:PRINT "Characters saved on file"
1640      c$=GET$
1650      IF INSTR(nondispcodes$,c$)=0 THEN redisplay=TRUE
1660    ENDPROC

1680    DEF FNhex(d)
1690      IF d>9 THEN =CHR$(ASC("A")+d-10)
                    ELSE =CHR$(ASC("0")+d)

1710    DEF PROCoutstring(c, s$)
1720    LOCAL i,l
1730      l = LEN(s$)
1740      FOR i = 1 TO l
1750        BPUT# c, ASC(MID$(s$,i,1))
1760      NEXT i
1770      BPUT# c,13
1780    ENDPROC
```

```
1800    DEF PROCrestore
1810    LOCAL r,c,sr,comma,char,byte,channel,fname$
1820      planning=FALSE
1830      PRINT:PRINT:PRINT
1840      INPUT "Filename",fname$
1850      PRINT:PRINT "Load tape and press PLAY"
1860      channel = OPENIN(fname$)
1870      FOR r=1 TO rows
1880        FOR c=1 TO cols
1890          FOR comma=1 TO 2
1900            REPEAT:char=BGET#channel
1910            UNTIL char=ASC(",")
1920          NEXT comma
1930          FOR sr = 1 TO 8
1940            char=BGET# channel
1950            char=BGET# channel
1960            byte=FNvalhex(char)*16
1970            char=BGET# channel
1980            byte=byte+FNvalhex(char)
1990            pic(c,r,sr)=byte
2000            char=BGET# channel
2010            PRINT ",&";~byte;
2020      NEXT:PRINT:NEXT:NEXT
2025      CLOSE# channel
2030      PRINT '"Characters restored."
2040      c$=GET$
2050      IF INSTR(nondispcodes$,c$)=0 THEN redisplay=TRUE
2060    ENDPROC

2080    DEF FNvalhex(d)
2090    IF d>ASC("9") THEN =d-ASC("A")+10 ELSE =d-ASC("0")

2120    DEF PROChelp
2130      PRINT:PRINT:PRINT
2140      PRINT "Cursor arrows : Move around grid"
2150      PRINT "F             : Foreground"
2160      PRINT "B             : Background"
2170      PRINT "V             : Display VDU codes"
2180      PRINT "S             : Save on file"
2190      PRINT "R             : Restore from file"
2200      PRINT "I             : Initialise"
2210      PRINT "H             : Help"
2220      PRINT "0-6           : Display picture in mode"
2230      PRINT "Q             : Quit"
2240      PRINT "Any other key : Planning grid"
2250      PRINT:PRINT
2260      PRINT "After quitting, saved VDU statements"
2270      PRINT "can be absorbed into a program by:"
2280      PRINT:PRINT "  *EXEC ""filename"" "
2290      c$=GET$
2300      IF INSTR(nondispcodes$,c$)=0 THEN redisplay=TRUE
2310    ENDPROC
```

Exercises

1 Define four different characters that represent the same
   spaceship in four different orientations. Incorporate
   these definitions in a program that moves the spaceship
   about the screen under the control of various keys that
   can be used to make it move forward, stop, turn right or
   turn left.

2 Design a block of four characters that represent a man
   with his arms outstretched and his legs apart. Define
   other characters that represent his arms and legs at
   different angles. Use your characters to display a 'flic-
   pic' by repeatedly displaying the man in the same place
   using randomly selected limb positions.
       Note that if the characters 224, 225, 226 are the
   codes for three different versions of his top left-hand
   quarter, you can make a random selection of one of these
   characters and display it by, for example:

       PRINT TAB(20,10); CHR$(223+RND(3));

3 Design blocks of characters that represent different
   types of vehicles in a mode of your choice.
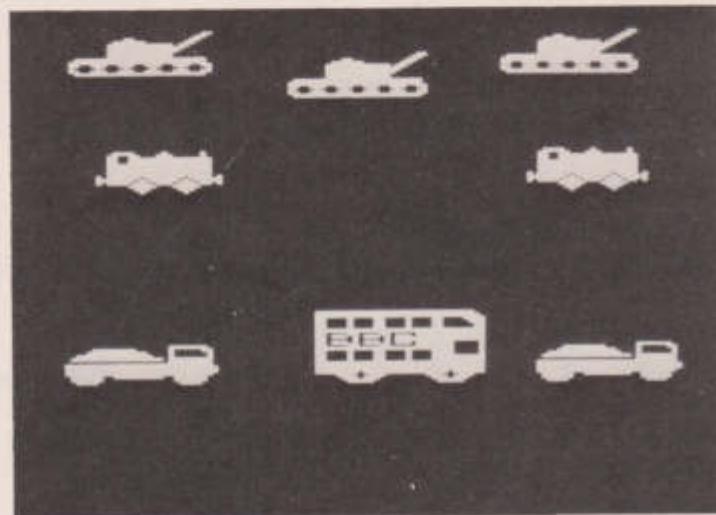
4 Design blocks of characters that represent different
   chess pieces.

5 Define a character that looks like an explosion and
   display this briefly when a spaceship is shot down, or
   when a spaceship collides with the gun. Make it flicker
   by overprinting it with a space and reprinting it several
   times.

6 In the 'bat'n'ball program, we used the underline
   character for constructing the bat. Because of this,

there is a gap between the bat and the ball when they meet. Design and use a character for the bat that will eliminate this difficulty.

11.6 Multi-frame images – refining character animation

We saw in the last section how we could construct a picture from a group of user-defined characters that were displayed together on the screen. It is possible to create interesting effects by defining more than one version of the same object, each one slightly different from the others. In Exercise 2 at the end of the last section, we saw how this technique could be used to create a man waving his arms and legs.

By defining several versions or 'frames' for an object that is moving around the screen, we can improve the animation in two possible ways. First of all, the technique can be used to produce smoother motion by making the object move in steps of less than one character. Secondly, parts of the object can be made to move independently of the overall motion of the object, for example arms and legs can be made to swing.

Smoother movement

We first show how we can make our car move across the screen in steps that are smaller than one character position. We shall define two frames or versions of our car as follows:
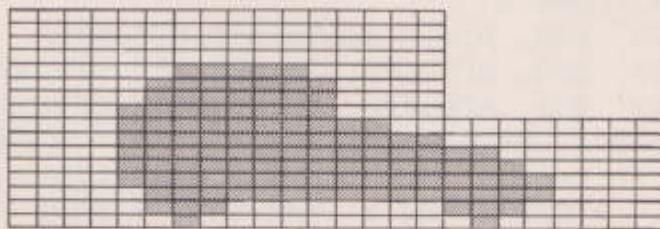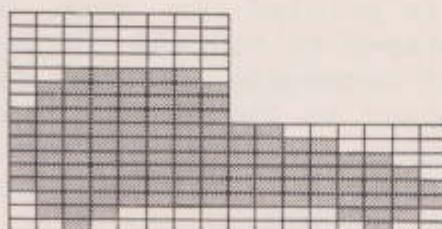
During the motion, these two frames will be displayed one after another in the same position, before stepping forward by one character position and displaying them again. First we define the string of characters that make up frame 1:

```
10       REM frame 1. Characters for top half of car.
20    VDU 23, 224, 0,0,0,0,3,7,7,&F
30    VDU 23, 225, 0,0,0,0,&E0,&F0,&F0,&F0
40       REM frame 1. Characters for bottom half of car.
50    VDU 23, 226, &F,&F,&F,&F,&F,7,7,2
60    VDU 23, 227, &FC,&FF,&FF,&FF,&FF,&FF,0,0
70    VDU 23, 228, 0,0,&C0,&E0,&F0,&F0,&E0,&40

80    bsp$ = CHR$(8)   :REM backspace
90    dn$  = CHR$(10) :REM down line
100   up$  = CHR$(11) :REM up line

110   frame1$=CHR$(17) + CHR$(1) + CHR$(224) + CHR$(225)+
                 bsp$ + bsp$ + dn$ + CHR$(17) + CHR$(2) +
                 CHR$(226) + CHR$(227) + CHR$(228) +
                 bsp$ + bsp$ + bsp$ + up$
```

CHR$(17) can be used to select a new colour for subsequent characters. Printing 'CHR$(17) + CHR$(1)' is equivalent to obeying the statement:

COLOUR 1

and printing 'CHR$(17) + CHR$(2)' is equivalent to obeying

COLOUR 2

Thus, whenever 'frame1$' is printed (in mode 5), the top half of the car is displayed in red and the bottom half in yellow. The combination of backspace, up-line and down-line characters has been chosen so that after printing frame1, the (invisible) text cursor is in position ready for printing frame 2. The program for moving the car across the screen can be completed as follows:

```
120      REM frame 2
130   VDU 23, 229, 0,0,0,0,&3E,&7F,&7F,&FF
140   VDU 23, 230, &FF,&FF,&FF,&FF,&FF,&7F,&70,&20
150   VDU 23, 231, &C0,&F0,&FC,&FE,&FF,&FF,&E,4
160   frame2$ = CHR$(17) + CHR$(1) + " " + CHR$(229) +
                 bsp$ + bsp$ + dn$ + CHR$(17) + CHR$(2) +
             " " + CHR$(230) + CHR$(231) + bsp$ + bsp$ + up$

200   MODE 5 : VDU 23; 8202; 0;0;0;
210   PRINT TAB(0,10); frame2$;
220   FOR x%=1 TO 17
230     PRINT frame1$;
240     PRINT frame2$;
250   NEXT x%
260   K = GET : MODE 7 : END
```

If you want to slow down the motion with a delay loop, remember that two delays will be necessary, one before printing frame 1 and one before printing frame 2.

Independent movement of parts
In the last section, the two image frames for the car differed only in the horizontal position of the car. We now show how slight changes in the shape of an object from frame to frame can be used to produce more effective animation. We shall illustrate this by animating an elephant that walks across the screen. We shall define only two frames for the elephant:
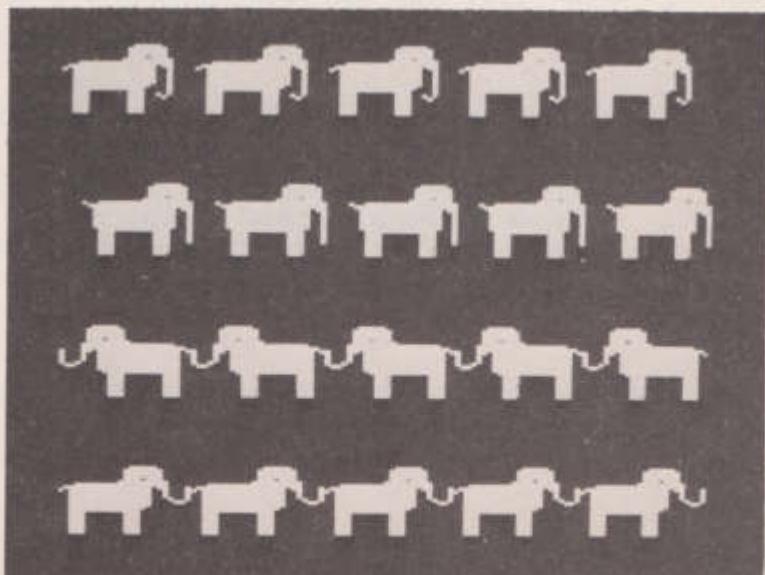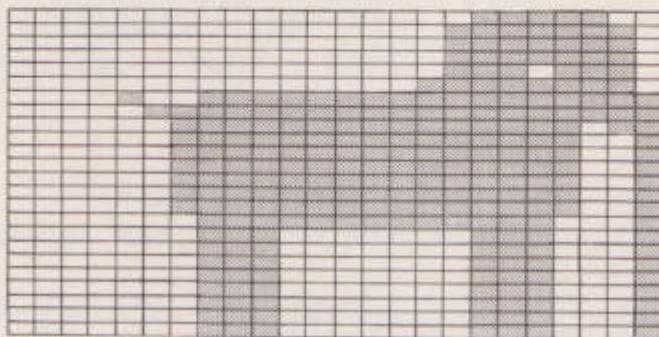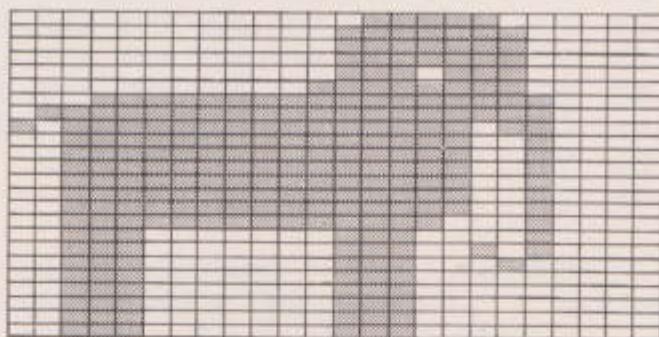
Not only does the horizontal position of the elephant change from frame to frame, but the positions of the legs, trunk and tail relative to the rest of the body also change. The photograph shows, in the first two rows, several copies of each of the above two frames in MODE 5. The next two rows show two more variations on the same theme. The two frames for the elephant are defined as follows:

```
  5           REM frame 1
 10   VDU 23, 224, 0,0,0,0,0,0,&1F,&7F
 20   VDU 23, 225, 7,&F,&F,&F,&E,&1F,&FF,&FF
 30   VDU 23, 226, &C0,&E0,&E0,&E0,&E0,&E0,&F0,&F0
 40   VDU 23, 227, &BF,&3F,&3F,&3F,&3F,&3F,&3F,&3F
 50   VDU 23, 228, &FF,&FF,&FF,&FF,&FF,&FF,&FF,&FF
 60   VDU 23, 229, &B0,&90,&90,&90,&90,&90,&90,&10
 70   VDU 23, 230, &38,&38,&38,&38,&38,&38,&38,&38
 80   VDU 23, 231, &E,&E,&E,&E,&E,&E,&E,&E
 90   VDU 23, 232, &10,&50,&20,0,0,0,0,0

100   bsp$ = CHR$(8) : dn$ = CHR$(10) : up$ = CHR$(11)
110   frame1$ = " " + CHR$(224) + CHR$(225) + CHR$(226) +
                bsp$ + bsp$ + bsp$ + bsp$ + dn$ +
                " " + CHR$(227) + CHR$(228) + CHR$(229) +
                bsp$ + bsp$ + bsp$ + bsp$ + dn$ +
                " " + CHR$(230) + CHR$(231) + CHR$(232) +
                bsp$ + bsp$ + bsp$ + up$ + up$

205           REM frame 2
210   VDU 23, 233, 0,0,0,0,0,0,9,7
220   VDU 23, 234, 0,0,0,0,0,1,&FF,&FF
230   VDU 23, 235, &7C,&FE,&FE,&FE,&EE,&FE,&FF,&FF
240   VDU 23, 236, 3,3,3,3,3,3,3,1
250   VDU 23, 237, &FF,&FF,&FF,&FF,&FF,&FF,&FF,&FF
260   VDU 23, 238, &FB,&F9,&F9,&F9,&F9,&F9,&F9,&F1
270   VDU 23, 239, 1,1,1,1,1,1,1,1,
280   VDU 23, 240, &C0,&C0,&C0,&C0,&C0,&C0,&C0,&C0
290   VDU 23, 241, &71,&71,&71,&71,&70,&70,&70,&70

310   frame2$ = CHR$(233) + CHR$(234) + CHR$(235) +
                bsp$ + bsp$ + bsp$ + dn$ +
                CHR$(236) + CHR$(237) + CHR$(238) +
                bsp$ + bsp$ + bsp$ + dn$ +
                CHR$(239) + CHR$(240) + CHR$(241) +
                bsp$ + bsp$ + bsp$ + up$ + up$
```

The loop for making the elephant walk across the screen is exactly the same as that used for the car in the last section. The effect produced by the program is rather more interesting.

Exercises

1 Make the car move in smaller steps by defining four frames that represent the car in four different horizontal positions.

2 Define several different frames to represent a man in different walking positions and write a program that makes the man walk across the screen.

3 Use four different character shapes for the spaceship in four different orientations and make the spaceships spin as they move down in the space-attacker program.

4 Make our elephant walk across the screen and point with its trunk to one of several words displayed in a column at the right of the screen.

5 Because of the low resolution in MODE 5, our elephant has rather square corners. Use more characters to redesign the elephant in a mode with greater resolution, such as MODE 4, and animate it as before.

11.7 Special effects with palette changing

We have already seen in Chapter 9 that the actual colour displayed on the screen for a particular colour code can be selected from any one of the sixteen available colours. In MODE 4, only two colours (code numbers 0 and 1) can be displayed on the screen at any one time, but these two colours can be any two of the sixteen available. Thus the spaceship and gun in our space-attacker program can be displayed in yellow instead of white if PROCinitialise includes:

```
135     VDU 19, 1,3, 0,0,0
```

This says that the colour with code number 1 in MODE 4 should be displayed as actual colour number 3 (yellow). All objects printed on the screen during the game will now be displayed in yellow.

Obeying a VDU 19 statement like this does not involve replotting information on the screen and the colour change is effected instantaneously by the display hardware during the next raster scan of the screen (which takes about one fiftieth of a second). We have already made use of this instantaneous colour change in Section 11.1 where setting a colour code to black allowed us to draw an object in that colour without the drawing process being seen. By switching to a foreground colour, the drawn object can be made to appear instantaneously. In the following sections, we demonstrate how this palette changing facility can be used

to create a number of special effects.

A gunflash effect

A VDU 19 command can be used to instantaneously change the background colour of the whole screen and this can be used to create a flash when a gun is fired, or for an explosion.

In the space-attacker program, we can change the background colour to red during the short period that it takes to draw and erase the bullet track. The background colour number is 0 and we need to insert a VDU 19 command to change the actual colour associated with code number 0 at the start of PROCfire:

```
725      VDU 19, 0,1, 0,0,0
```

We need to reset colour number 0 to actual colour 0 (black) at the end of PROCfire:

```
755      VDU 19, 0,0, 0,0,0
```

A spinning disc

The following program draws a circular disc consisting of 30 different segments, where consecutive segments are plotted using colour codes 1,2,3,1,2,3, etc. In MODE 5, colour code number 2 is initially associated with actual colour 3 (yellow) and colour code number 3 is initially associated with actual colour 7 (white). The VDU 19 statements at the start of the program simply tidy this situation up by ensuring that colour code numbers 1,2,3 are associated with actual colours 1,2,3 (red, green, yellow).

```
10      MODE 5
20      VDU 19, 2,2, 0,0,0
30      VDU 19, 3,3, 0,0,0
40      VDU 29, 600;500;    :REM move graphics origin
50      r = 300    :REM radius of disc
60      segs = 30  :REM number of segments
70      theta = 0 : thetainc = 2*PI/segs
                            :REM angle and increment
80      MOVE r,0
90      FOR seg = 1 TO segs
100        theta = theta + thetainc
110        x = r*COS(theta) : y = r*SIN(theta)
120        GCOL 0, seg MOD 3 + 1
130        MOVE 0,0 :REM establish triangle for colour fill
140        PLOT 85, x,y    :REM colour fill segment
150      NEXT seg
```
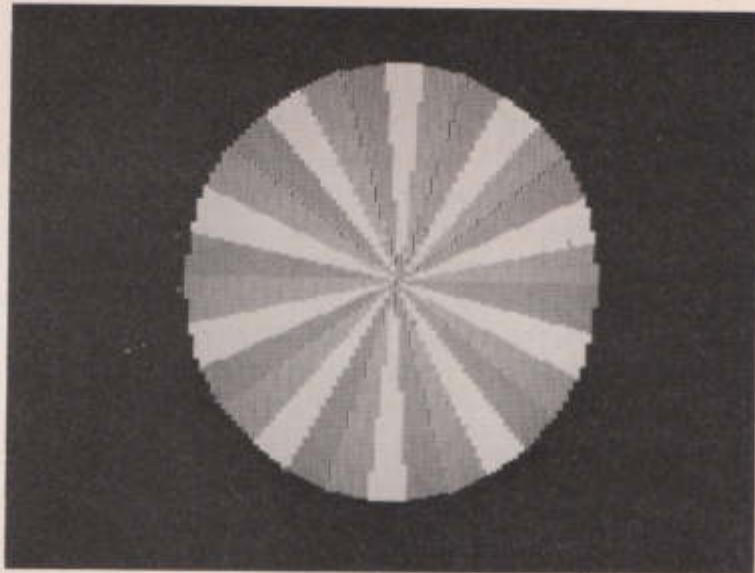
By repeatedly changing the actual colour of each segment to the actual colour of an adjacent segment, we can create the illusion that the disc is spinning. One way of doing this is to extend the program as follows:

```
310     displacement = 0
320     REPEAT
330       FOR code = 1 TO 3
340         actual = (code + displacement) MOD 3 + 1
350         VDU 19, code,actual, 0,0,0
360       NEXT code
370       displacement = (displacement+1) MOD 3
380     UNTIL INKEY$(0)=" "
385
390     MODE 7 : END
```

The loop from line 330 to line 360 changes the actual colours associated with codes 1,2,3 to the next sequence of values required. You should tabulate the values of the various arithmetic expressions involved in the above process to see how it works. The speed of rotation of the disc can be slowed down by increasing the parameter of INKEY$ at line 380.

Spiral patterns

Some interesting effects can be obtained by replacing the disc of the previous section by various spiral patterns and using palette changing to make these rotate. First of all, we draw a band of colour that spirals outwards from the centre of the screen. Consecutive segments of the spiral are plotted in different colours.

```
10      MODE 5 :REM use MODE 1 if you have a Model B
20      VDU 19, 2,2, 0,0,0
30      VDU 19, 3,3, 0,0,0
40      width = 20   :REM width of band of colour
50      segs = 20    :REM coloured segments per rotation
60      layers = 15 :REM complete rotations in spiral
70      r = 0 :REM radius that increases as spiral is drawn
80      theta = 0 :REM angle used to sweep out the spiral
90      rinc = width/segs :REM radius increase per seg
100     thetainc = 2*PI/segs :REM theta increase per seg
110     VDU 29, 600;500; :REM move graphics origin
120     x1 = 0 : y1 = 0 : x2 = width : y2 = 0
130     MOVE x1,y1

140     FOR step = 1 TO segs*layers
150        GCOL 0, step MOD 3 + 1
160        r = r + rinc : theta = theta + thetainc
170        newx1 = r*COS(theta) : newy1 = r*SIN(theta)
180        newx2 = (r + width)*COS(theta)
190        newy2 = (r + width)*SIN(theta)
200        MOVE x2,y2
210        PLOT 85, newx2,newy2
220        MOVE x1,y1
230        PLOT 85, newx1,newy1
240        x1 = newx1 : y1 = newy1
250        x2 = newx2 : y2 = newy2
260     NEXT step
```

You should run the above program with different values for the variable 'segs'. If the number of segments is an exact multiple of the number of colours being used (3 in this case), then colours in adjacent layers of the spiral will match and the effect obtained is similar to that of the disc of the last section. If the number of segments is a whole number that is not a multiple of 3, then the colours in adjacent layers will not match and the spiral effect is more evident. The best effects are obtained by setting 'segs' to a non-integer value that is close to a multiple of the number of colours used. Try, for example,

```
50      segs = 20.5
```

Interesting effects can also be obtained with small values of 'segs'. Try 3.5, 4, 4.5, 5, 5.5.
Now add lines 310 onwards from the rotating disc program to make the spiral rotate. The effectiveness of the illusion will depend on the value used for 'seg'. You can again vary the speed of rotation by adjusting the parameter of INKEY$.

segs = 20

segs = 20.5

segs = 4.5
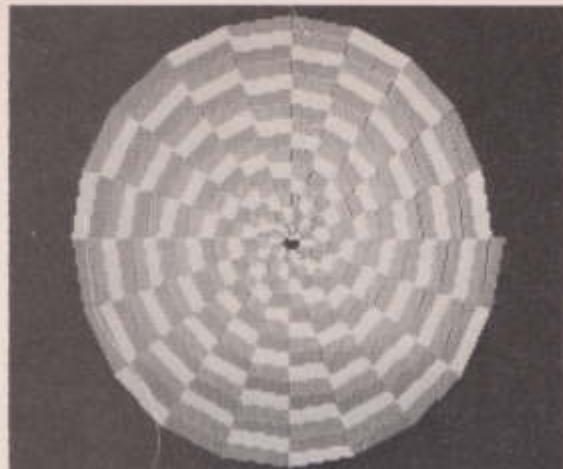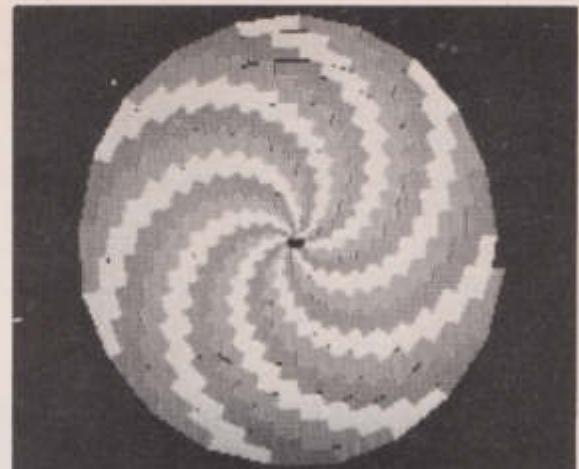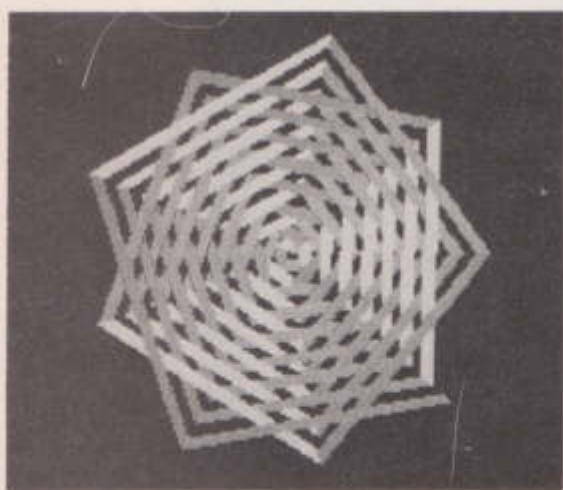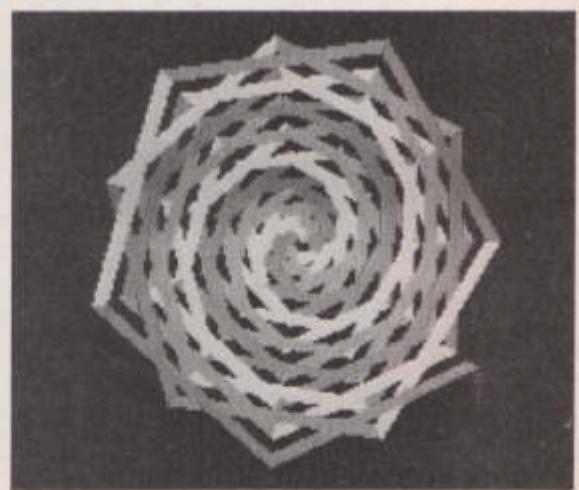
segs = 5.5

We can obtain a catherine wheel effect by replacing each segment of solid colour by a collection of coloured spots. One way of doing this is to replace lines 160 to 250 by

```
160        FOR spot = 1 TO spots
170          rp = r + RND(1)*width
180          thetap = theta + RND(1)*thetainc
190          x = rp*COS(thetap)
200          y = rp*SIN(thetap)
210          PLOT 69, x, y
220        NEXT spot
230
240        r = r + rinc
250        theta = theta + thetainc
```

where 'spots' is initialised to a small value such as 10. Although the catherine wheel spins very quickly once it has been drawn, you will find that it takes several minutes to draw. The reason for this is that SIN and COS are very time-consuming functions. It is possible to generate a random set

of points within a given segment without using SIN and COS, but we leave this as an exercise for those proficient in coordinate geometry. A better catherine wheel effect is obtained if one of the colours used in the palette-changing process is the background colour. This can be done by changing the calculation at line 340 or by using the technique suggested in Exercise 1 below. When we used black (background), red and yellow as our colours, the firework looked like this.
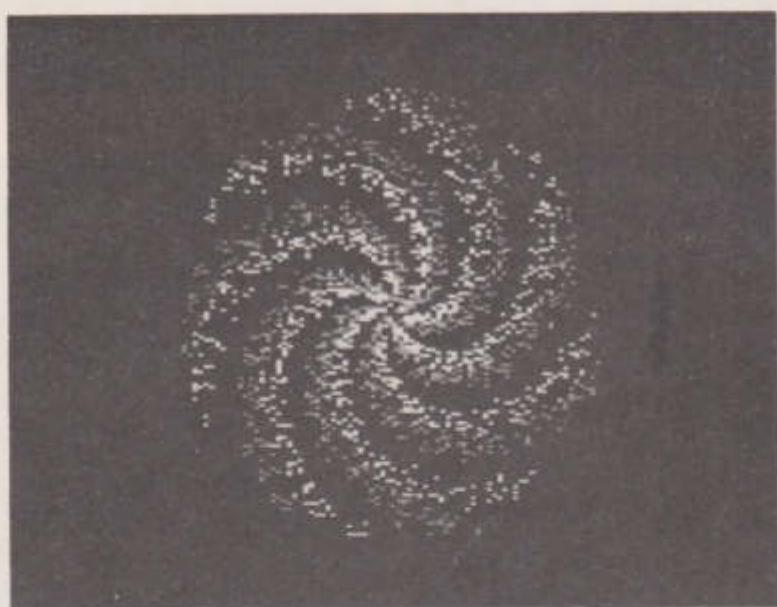
Exercises

1   At line 340 of the palette-changing process used in the last two sections, an actual colour (1, 2 or 3) was selected by calculation. By storing three actual colour numbers in locations 1, 2 and 3 of an array, and using the number calculated at line 340 as a subscript for this array, any combination of three colours from the sixteen available can be selected. Use this technique to experiment with different combinations of actual colours in the above programs.

2   If you have access to a Model B computer, modify the palette-changing programs to run in MODE 2 with a bigger range of colours.

11.8 Joining the graphics and text cursors

One disadvantage of the character printing techniques that we have used so far for creating animation effects is that when a character shape is printed it completely obliterates anything that previously appeared on the screen in that area. None of the logical plotting operations that are

selected by GCOL statements have any effect on the way that characters are normally displayed on the screen. This means that, when using normal printing techniques, we can not create the special effects such as foreground and background image planes that were illustrated in Chapter 9.

In this section we introduce the VDU 5 statement which 'joins' the graphics and text cursors. Issuing this command has a number of consequences:

(a) The colour for characters printed is selected by GCOL statements. This means that characters can be inserted on the screen under the control of the various logical operations on colour codes. We can create the effect of several image planes while still retaining the advantages of user-defined characters.

(b) A character can be displayed starting anywhere in the current graphics area by preceding the PRINT statement by a MOVE statement. The top left hand point of the character is displayed in the pixel at the current graphics position. This is useful for annotating graphs and charts. After printing a character, the graphics position is at the top left hand corner of the area where the next character would normally be printed. Using TAB also moves both cursors.

(c) Only the foreground part of the character is plotted. Printing a space on top of a previously displayed character no longer has the effect of erasing it. This means that different shapes can be overlayed under the control of different GCOL statements and we are no longer limited to a single foreground colour within the area of a given character.

(d) When printing single characters, full windowing of the screen is effective. If a character is 'printed' off screen, it does not automatically appear elsewhere on the screen. Nor does any scrolling take place. However, if a string containing more than one character is printed at the edge of the screen, strange effects can take place.

The price we pay for these advantages is loss of speed. It takes much longer to print a character after a VDU 5 statement has been obeyed. This is because of all the extra work that is being done behind the scenes in applying logical operations to colour codes and in making windowing calculations. You should therefore bear in mind that, despite the exciting possibilities, speed can be a problem when complex animation is involved. For example, when multi-character shapes such as the car or the elephant are being animated, printing takes place so slowly that the flashing effect observed in Section 11.1 returns and we would have to

use the alternate image plane technique of section 11.1 for satisfactory results.

The following program introduces some of the effects that can be created by using the VDU 5 statement.

```
10      MODE 5
20      *FX 4,1
30      *FX 11,5
40      *FX 12,5
50      VDU 5
60      VDU 23, 224, &18,&18,&18,&3C,&7E,&C3,&7E,&3C
70      VDU 23, 225, 0,0,0,0,0,&3C,0,0
80      VDU 19, 2,3, 0,0,0
90      VDU 19, 3,1, 0,0,0
100     ship$  = CHR$(18) + CHR$(3) + CHR$(1) + CHR$(224)
110     light$ = CHR$(18) + CHR$(3) + CHR$(2) + CHR$(225)

120     GCOL 0,2
130     FOR star = 1 TO 500
140       PLOT 69, RND(1279), RND(1023)
150     NEXT star

160     x% = 500 : y% = 500
170     step% = 64
180     PROCship(x%, y%)

190     REPEAT
200       k% = GET
210       ON k% - 135 GOTO 211, 212, 213, 214  ELSE  250
211         nx% = x% - step% : ny% = y% : GOTO 220
212         nx% = x% + step% : ny% = y% : GOTO 220
213         nx% = x% : ny% = y% - step% : GOTO 220
214         nx% = x% : ny% = y% + step% : GOTO 220
220       PROCship(x%, y%)
230       PROCship(nx%, ny%)
240       x% = nx% : y% = ny%
250     UNTIL k% = 32   : REM 32 is code for a space

260     *FX 4,0
270     *FX 12,0
280     MODE 7 : END

310     DEF PROCship(x%, y%)
320       MOVE x%, y%
330       PRINT ship$
340       MOVE x%,y%
350       PRINT light$
360     ENDPROC
```

We use the same 'spaceship' character as before, but this time we define a second character to fill the 'hole' in the

spaceship with a different colour so that the hole now looks like a light. We also use the foreground-midground technique introduced in Chapter 9 so that our spaceship moves against a background of stars (single pixels lit up using PLOT 69). The stars are obscured when the spaceship passes over them. The movement of the spaceship is controlled by the user who moves it by pressing the four arrow keys.
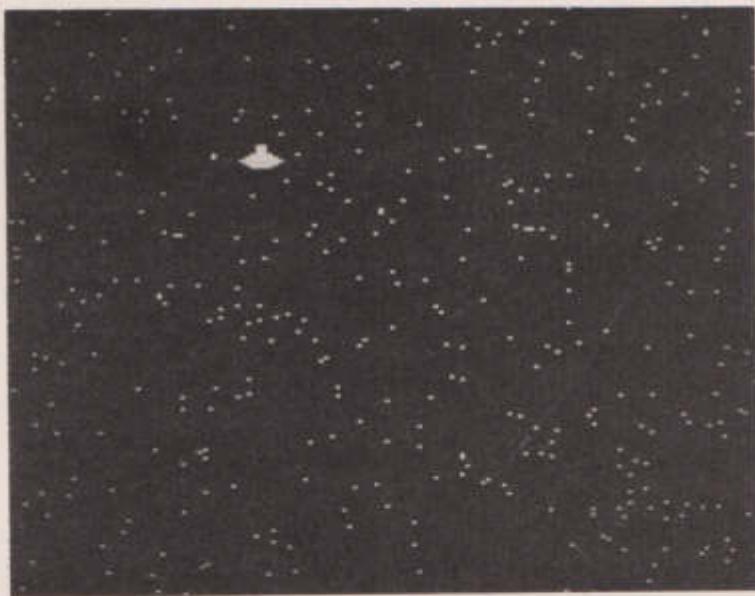
Because of a shortage of colour codes in MODE 5, we have 'cheated' and the light on the spaceship is plotted in the same colour as the stars. This has an odd effect if the light passes over a star, but does not do much harm. The various possible colour codes at a pixel and their significance are as follows:

Colour code	Bit pattern	Significance
0	00	(black) sky
1	01	(red) spaceship obscuring sky
2	10	(yellow) star or light
3	11	(red) spaceship obscuring star

Note that the character whose code is 18 can be used to create the effect of a GCOL statement. For example, printing the string

    CHR$(18) + CHR$(3) + CHR$(1)

has exactly the same effect as the statement

    GCOL 3, 1

Finally, note that the VDU 5 effect is switched off by VDU 4.

Exercises

1. Interesting effects can be obtained when the spaceship passes over stars if colour code 3 is set to one of the flashing colours. Try this.

2. Modify the space attacker program to use the VDU 5 statement. There are a number of alterations that will need to be made. The simplest alteration to make is to issue the instruction

   GCOL 3,1

   in PROCinitialise and replace all the spaces previously used to erase characters by copies of the characters themselves. After these alterations have been made, you will find that the last character of the bullet trace no longer erases a spaceship that has been shot down and this will have to be dealt with. Firing a bullet will now be a much slower process and a faster alternative would be to draw and immediately erase (by drawing again) a line running up the screen from the gun.

3. Now try plotting a background of stars for the space attacker program. Although we are working in MODE 4 with only two colours available, you should find that a spaceship can pass over the stars without erasing them. (Actually, a hole appears momentarily in the spaceship as it passes over a star.) This is because all our plotting and erasing is being done using the exclusive or option.

4. Modify the car programs written earlier to run with the VDU 5 option. Observe the problems that arise and try to cure them. Try making the wheels a different colour from the body by overlaying different characters in different colours.

Appendix 1 Typing and editing programs

The keyboard – general

Most of the keys on the keyboard, when pressed, cause characters to appear on the screen. These are the letter keys, each of which is marked with a letter of the alphabet, and the symbol keys, each of which is marked with two symbols. In addition, there are a number of keys that are used for special purposes.

The keyboard can be in one of three states:

(a) CAPS LOCK on: This is indicated by one of the red lights at the front of the keyboard. The computer is in this state when first switched on. In this state the letter keys can only be used for typing capital letters. The symbol keys behave as they do on a normal typewriter: simply press the key to type the lower of the two symbols on the key; to type the upper of the two symbols on a key hold down the SHIFT key while you press the symbol key. CAPS LOCK is switched on or off by pressing the CAPS LOCK key once.

(b) CAPS LOCK off (and SHIFT LOCK off): In this state, the letter keys normally type small letters. To type a capital letter, hold down the SHIFT key and press the letter key. The symbol keys behave normally.

(c) SHIFT LOCK on: This is indicated by another of the red lights at the front of the keyboard. In this state, only capital letters and the upper symbol on each of the symbol keys can be typed.

If you adopt the typing style used for programs in this book, you will find it most convenient to switch the CAPS LOCK off while typing BASIC programs (holding down the SHIFT key while typing BASIC keywords). While running, listing, saving or loading programs, it is best to switch CAPS LOCK on. You are unlikely to need SHIFT LOCK.

The RETURN key

The computer will not pay attention to anything you type until you press the RETURN key. This key must be pressed after each command and after each line of BASIC that you type.

The DELETE key

Before pressing RETURN, you can edit the line currently being typed by using the DELETE key to delete characters.

The ESCAPE key

This is used to tell the computer to interrupt whatever it is doing and await new instructions. It does not delete your current program.

The BREAK key

This has an effect similar to, but more drastic than, the ESCAPE key. It switches the machine to MODE 7 (clearing the screen) and wipes out the current program (although this can be restored using the OLD command).

The COPY key and cursor arrows

These are used for editing programs - see below.

The user-defined function keys, F0 to F9

The effect of these keys can be defined by using the *KEY command. This command is followed by the number of the key that you want to define and a string that contains the commands that are to be obeyed when the key is pressed. For example

*KEY 0, "PRINT"   Every time F0 is pressed, it
                               behaves as if the word PRINT
                               had been typed.

*KEY 9, "MODE 7:LIST |M"   Every time F9 is pressed, the
                               machine will switch to MODE 7
                               and the current program will
                               be listed. The |M (vertical
                               bar followed by M) is used in
                               the string in place of RETURN.

In fact the effect of other keys can be defined in this way. BREAK is F10, and in later versions of the operating system the command *FX 4,2 will permit COPY and the cursor arrows to be redefined as F11 to F15.

Program line numbering

Lines are usually numbered in steps of 10. This makes it possible to insert new lines between the existing lines of a program by giving numbers that fall between the existing line numbers.

In the text, we have often left larger gaps in the

numbering, for example we have sometimes started numbering each procedure with a different multiple of 100. This makes it easier to present later extensions to programs where the new lines are still numbered in steps of 10.

If necessary, the lines of a program can be renumbered by using the command RENUMBER. When used on its own, this command renumbers the lines of the current program starting with line 10 and going up in steps of 10. If RENUMBER is followed by a single parameter, that parameter specifies the number to be used for the first line of the program and the remaining lines are numbered in steps of 10. If RENUMBER is followed by two parameters, then the first parameter specifies the number for the first line of the program and the remaining line numbers go up in steps that are specified by the second parameter. For example:

```
RENUMBER 500        Lines are numbered 500, 510, 520, 530, ...

RENUMBER 200,5      Lines are numbered 200, 205, 210, 215, ...
```

Listing programs

A complete program can be listed on the screen by using the LIST command. If a program is too long to fit on the screen, then part of the program can be listed by giving line numbers in the LIST command, for example:

```
LIST 360            List line 360 only.

LIST 400,550        List lines from 400 to 550 inclusive.

LIST 460,           List lines from 460 to the end.

LIST ,200           List lines up to line 200.
```

Deleting lines

To delete a single line from a program, type the line number followed immediately by RETURN. To delete several lines from a program, use the DELETE command, for example:

```
DELETE 320,460      Deletes lines 320 to 460 inclusive.
```

Blank lines

It improves the readability of a program if blank lines are occasionally inserted between different program sections that carry out logically separate tasks. There are two ways of doing this. A blank line can be forced after a given line by typing extra spaces at the end of the line so that the

line overflows onto the next screen line. Alternatively, blank lines can be numbered, but the number must be followed by at least one space.

Long lines

A single numbered line in a BASIC program can occupy up to 240 characters, or six screen lines in MODE 7. When a long line is being typed, you must not press RETURN until the complete BASIC line has been typed. The BASIC line will automatically overflow onto the next screen line when necessary.

Spaces and indentation

If speed of program execution is not critical (it rarely is) and program storage space is not in short supply, then extensive use of extra spaces typed in a program can help to make it much easier to read. At least one space should be inserted after each line number and spaces should be used to separate BASIC keywords from variable names. Sections of program can be made to stand out by 'indenting' them (typing one or two extra spaces at the start of each line) and this should be done between FOR and NEXT, between REPEAT and UNTIL, and in the definitions of procedures and functions. The layout of a long line can often be improved by forcing overflow onto the next screen line at a sensible point by typing extra spaces.

Omitting such redundant spaces makes a <u>minute</u> difference to the speed of execution of a program, and in the <u>rare</u> cases where speed of execution is critical, these <u>redundant</u> spaces can be omitted.

Extra spaces also take up memory space, and they may occasionally have to be omitted in order to fit a very large program into the computer.

If extra spaces have to be omitted, the LISTO (LIST Option) command can be used to slightly improve the layout of programs when listed. For example,

LISTO 7

Says that an extra space should be inserted after each line number and FOR and REPEAT loops should be indented by two extra spaces whenever a program is LISTed. This command will not do anything to improve the layout of multi-line statements.

Cursor editing

The flashing 'cursor', in the shape of an underline character, normally indicates where on the screen the next character typed from the keyboard (or printed by the computer) will appear. The four arrow keys at the top right-

303

hand corner of the keyboard can be used to change a line of
program without having to retype it. Pressing one of the
arrows immediately creates two cursors: the white block
marks the point on the screen where the next character of
the new line being typed will appear. The flashing underline
symbol is the 'editing cursor' or 'copy cursor'. This cursor
can be moved about the screen by using the arrows and any
character currently marked by the copy cursor can be copied
into the new line by pressing the COPY key. The computer is
said to be in 'edit mode'.

Editing a line of program

To edit a line of the current program, you must first LIST
it on the screen, then use the arrows to move the copy
cursor to the start of the displayed line that is to be
changed. Now use the COPY key to copy the parts of the old
line that are correct, and type any additions or corrections
from the keyboard. The arrow keys can be used to skip parts
of the old line that are to be omitted, or even to move to
another line and include part of that in the new line being
constructed. Mistakes in the new line can be corrected with
the DELETE key in the normal way and when the new line is
complete, the RETURN key should be pressed.

Appendix 2 **Cassette files**

Saving programs on file

A BASIC program can be saved on cassette by using the SAVE command, for example

    SAVE "progname"

where the program name can have up to ten characters. The computer will tell you to press RECORD on the tape recorder and then press RETURN. (The tape recorder must, of course, be connected to the socket marked 'cassette' at the back of the computer.) The program is recorded on the cassette in the form of a 'file'. The file consists of a sequence of 'blocks' on which audio signals representing the program are recorded. As the program is being recorded, these blocks are counted (in hex) on the screen. In the gaps between the blocks, a pure audio tone is recorded. You can hear what the program 'sounds' like by playing back the tape. Incidentally, a program does not have to be complete before it is saved. You can save a large program when you are half way through typing it.

Loading programs from file

In order to reload a previously saved program into the computer, use the command

    LOAD "progname"

and press PLAY on the tape recorder. The blocks will again be counted on the screen as they are read by the computer. If the method used to connect your tape recorder to the computer does not automatically switch off the loudspeaker on the tape recorder, then you will find the noise made by the recorded programs disturbing. You have to insert a blank plug into the earphone socket in order to switch off the loudspeaker. (Do not use the short-circuit plug that is sometimes provided for insertion in the microphone socket when erasing tapes.)

You will have to experiment with the volume control on the tape recorder to find the best setting for reloading programs. The tone control should be adjusted to its highest setting. If the computer misreads one of the blocks, it will ask you to rewind the tape. (You do not need to rewind to the start of the program, only to the block that was misread.)

You may have problems loading a program that was saved with a different tape recorder. This can be caused by slight differences in tape head alignment or by differing characteristics of the record and playback circuits.

Cataloguing the programs on a tape

The *CAT command can be used to obtain a list of the names of the programs that are stored on a tape. Type this command and then play the tape. When you have obtained the information that you want, press ESCAPE.

If you save a long program that you have just typed, it is a good idea to rewind the tape and do a *CAT to make sure that the program has been recorded correctly before switching off the computer and losing the program.

Merging two programs

You may occasionally want to combine two BASIC programs into one. For example, you may have a collection of procedures developed in one program that you want to add to another program so that they can be called by the other program. You must first load one of the programs (usually the shorter) and renumber the lines so that they are different from the line numbers in the other program. For example,

    RENUMBER 2000

will renumber the lines of a program from 2000 upwards. The renumbered program should now be saved on cassette in character form. This can be done by using the *SPOOL command, for example,

    *SPOOL "ONEPROG"

After this command has been issued, everything that appears on the screen is also output in character form to the file "ONEPROG". To store our program or cassette in character form, we need only type

    LIST

The characters listed on the screen will also be sent to the file. If only part of the program is to be merged with the other one, then only selected lines need be listed. Output to the file is then terminated by the command

    *SPOOL

with no file name. If you do not have the remote control switch on your tape recorder connected, the computer will be unable to switch off the tape recorder automatically and there will longer gaps than usual between blocks. However,

this does not matter.

The other program must now be loaded in the usual way and the program that was saved in character form can now be added to the first by issuing the command

```
*EXEC "ONEPROG"
```

This tells the computer to read characters from the file "ONEPROG" and 'pretend' that they are being typed at the keyboard. This will have the effect of adding the previously renumbered program to the one that is currently in store.

Storing data on cassette files

There are many circumstances in which it is convenient to store information, such as numbers or strings, on files. If several programs require to process the same values, these values can be stored on a file and input from the file to each program as required. If a program is required to use the same large quantity of data each time it is run, there may not be room in store for these values to be held in DATA statements, and the solution would again be to create a file containing the values required by the program. Another possibility is that output sent to a file by one program can be supplied as input to a second program for further processing. A final important possibility is that a collection of data about some topic (a 'database') may be built up over a period by several runs of a program. The information could be held on a file between runs of the program. Each time it was run, the first thing the program would do would be to input the file containing the latest state of the database, and the last thing it would do would be to output the new state of the database to a new file. Perhaps we should mention that in most realistic database applications, the database is so large that it will not fit in the computer's main memory, and the files containing the database (usually disc files) are organised in such a way that part of the database can be input from file, updated, and output to file, as required.

Creating a file of numbers

To create a file, a program must do three things. It must tell the computer that it is about to create a file (i.e. it must 'open' the file by using the OPENOUT function). It must then send information to the file, in much the same way as information is sent to the screen, by using a variation of the PRINT statement called PRINT# (or by using BPUT# which is not covered here). Finally, the computer must be told that no more information is going to be sent to the file — the file must be 'closed' using the CLOSE# statement. For example, the following program allows the user to input a specified number of values (experimental readings, say) and

saves these values on a file.

```
10      fileno = OPENOUT("readings")
20      INPUT "How many readings", n
30      PRINT# fileno, n
40      FOR i=1 TO n
50        INPUT nextreading
60        PRINT# fileno, nextreading
70      NEXT i
80      CLOSE# fileno
```

Note that OPENOUT creates a file number which must be used subsequently by the program when using PRINT# to send information to the file and later when using CLOSE# to close the file. The only point at which the name of the file is mentioned is in the OPENOUT function.

The program could let the user choose his own name for the file of readings as follows:

```
5       INPUT "Name of file", filename$
10      fileno = OPENOUT(filename$)
```

Reading from a file of numbers

A file created by the above program could be read and the average of the readings in the file calculated as follows:

```
10      fileno = OPENIN("readings")
20      INPUT# fileno, n
30      sum = 0
40      FOR i = 1 TO n
50        INPUT# fileno, nextreading
60        sum = sum + nextreading
70      NEXT i
80      PRINT "Average: "; sum/n
90      CLOSE# fileno
```

If a file did not include an integer at the start indicating the number of values in the rest of the file, the end of the file could be recognised by using the EOF# function. For example, the average of the numbers in a file with no count at the start could be calculated by

```
10      fileno = OPENIN("readings")
20      sum = 0   : n = 0
30      REPEAT
40        INPUT# fileno, nextreading
50        n = n+1
60        sum = sum + nextnumber
70      UNTIL EOF#(fileno)
80      CLOSE# fileno
90      PRINT "Average: "; sum/n
```

Files containing strings

Strings can also be held as items in a file. For example, the following program sets up a file of strings that represent a piece of text typed at the keyboard.

```
10      fileno = OPENOUT("text")
20      INPUT LINE nextline$
30      REPEAT
40        PRINT# fileno, nextline$
50        INPUT LINE nextline$
60      UNTIL nextline$="****"
70      CLOSE# fileno
```

The user simply types the text a line at a time and types a line containing four stars to terminate the file. The four stars are not sent to the file and any program that reads from the file must use EOF# to recognise the end of the file.

Note that items are held in the file in the same way that they are held in the computer store. This means that each item in the file can only be input to a variable of appropriate type. In particular, if a string has been sent to a file by a statement such as

PRINT# fileno, "1, 2, 3"

then it can not subsequently be input by a statement such as

INPUT# fileno, x, y, z

It can only be input as a complete string, for example

INPUT# fileno, s$

Keeping a parallel array database on file

The use of parallel arrays for representing a simple in-store database was described in Chapters 6 and 7. For example, a simple stock list could be held in arrays declared by

    DIM refno(200), name$(200), stock(200), priceof(200)

These could be initialised from a file by a call of the procedure

```
100     DEF PROCinitialise
110     LOCAL stockfileno, i
120       stockfileno = OPENIN("stockfile")
130       INPUT# stockfileno, sizeofstock
140       FOR i = 1 TO sizeofstock
150         INPUT# stockfileno, refno(i), name$(i),
                                stock(i), priceof(i)
160       NEXT i
170       CLOSE# stockfileno
180     ENDPROC
```

At the end of a run of the program, an updated file could be created by a call of

```
200     DEF PROCcreatenewstockfile
210     LOCAL stockfileno, i
220       stockfileno = OPENOUT("stockfile")
230       PRINT# stockfileno, sizeofstock
240       FOR i = 1 TO sizeofstock
250         PRINT# stockfileno, refno(i), name$(i),
                                stock(i), priceof(i)
260       NEXT i
270       CLOSE# stockfileno
280     ENDPROC
```

Appendix 3 **Operator precedence**

Precedence level 1

    + applied to a single operand (unary plus)
    - applied to a single operand (unary minus)
    NOT
    function calls
    bracketed subexpressions
    indirection operators ?, !, $ (not covered)

Precedence level 2

    ^   raised to the power of

Precedence level 3

    *   times
    /   divided by
    DIV   divided by (on integers producing integer)
    MOD   remainder on dividing two integers

Precedence level 4

    +   plus
    -      minus

Precedence level 5

    =   equals
    <>   is not equal to
    <   is less than
    >   is greater than
    <=   is less than or equal to
    >=   is greater than or equal to

Precedence level 6

    AND   logical and (on truth values or bit patterns)

Precedence level 7

    OR   logical or (on truth values or bit patterns)
    EOR   exclusive or (on truth values or bit patterns)

Adjacent operators at the same precedence level are dealt with from left to right.

Appendix 4 **Summary of mode and colour facilities**

Text facilities available in different modes

mode	colours available	characters per line	lines
0	2	80	32
1	4	40	32
2	16	20	32
3	2	80	25
4	2	40	32
5	4	20	32
6	2	40	25
7	Teletext display	40	25

Graphics facilities available in different modes

mode	colours available	graphics resolution
0	2	640 x 256
1	4	320 x 256
2	16	160 x 256
4	2	320 x 256
5	4	160 x 256

Note that there no graphics facilities in modes 3, 6 and 7.

Memory requirements for different modes

mode	memory requirements
0	20K
1	20K
2	20K
3	16K
4	10K
5	10K
6	8K
7	1K

Overall colour range

There are sixteen actual colours available (on the Model A or B). These colours are numbered from 0 to 15.

Actual colour numbers and corresponding colours

colour number	colour name
0	black
1	red
2	green
3	yellow
4	blue
5	magenta
6	cyan
7	white
8	flashing black-white
9	flashing red-cyan
10	flashing green-magenta
11	flashing yellow-blue
12	flashing blue-yellow
13	flashing magenta-green
14	flashing cyan-red
15	flashing white-black

Colour codes in different modes

In each mode colours are referred to by code numbers from 0 upwards (using COLOUR for text colour and GCOL for graphics colour). The background colour is set by adding 128 to the required code number. The code numbers for a mode can be made to refer to any combination of actual colours (using VDU 19). There is an initial or default setting for each mode which specifies the colour that you get if you do not use VDU 19.

2 colour mode (MODES 0,3,4,6)

colour code numbers		default actual colours	
foreground	background	colour	number
0	128	black	0
1	129	white	7

4 colour mode (MODES 1 and 5)

colour code numbers		default actual colours	
foreground	background	colour	number
0	128	black	0
1	129	red	1
2	130	yellow	3
3	131	white	7

In the 16 colour mode (MODE 2) the colour codes are
initially set to the corresponding actual colour numbers.

Appendix 5 **Bits, bytes and hex**

For the majority of straightforward programming applications, the user of the BBC micro need not concern himself with the details of how things like numbers and strings are represented inside his computer, but for some advanced applications a more detailed knowledge of the internal representation of information is required.

Bits

All information stored in a modern digital computer is held in the form of 'binary digits'. In this context, the word 'binary' means 'having two possible values', and a binary digit can thus be set to one of two possible values. We usually abbreviate the term binary digit to 'bit'.

When we write a bit on paper, we represent its two possible values as 0 or 1. Inside a computer, a bit might be represented by a magnetic field lying in one of two possible directions, or by an electronic voltage that can be positive or negative. The programmer, however, need not concern himself with the practicalities of representing a bit electronically or magnetically. When he needs to think in terms of the binary representation of information, he can think entirely in terms of ones and zeros.

With one bit, we can represent only two possible values, 0 or 1, and in fact some of the information in our BBC computer is coded using only one bit. For example, in MODE 4, one bit is used to code the colour of each pixel on the screen. Each pixel can be one of two colours, colour 0 or colour 1.

Bit patterns

Bits are usually organised into groups or 'patterns'. With a group of two bits, each bit can one of two values giving 2x2 possible different patterns.

first bit	second bit	bit pattern
0	0	00
0	1	01
1	0	10
1	1	11

A two-bit pattern is used to code the colour of each pixel on the screen in a four colour mode such as MODE 5.

With three bits, there are 2x2x2 possible different patterns and so on:

314

no. of bits in pattern	example	no. of possible different patterns
1	0	2
2	10	4 = 2x2
3	011	8 = 2x2x2
4	1010	16 = 2x2x2x2
5	10100	32 = 2x2x2x2x2
6	011010	64 = 2x2x2x2x2x2
7	1101001	128 = 2x2x2x2x2x2x2
8	11000101	256 = 2x2x2x2x2x2x2x2

Bit numbering

The bits in a bit pattern are usually referred to by numbering them from zero upwards from right to left, bit0, bit1, bit2 and so on.

$$\ldots\ 1\ 0\ 1\ 1\ 0\ 1$$

```
...   bit5  bit4  bit3  bit2  bit1  bit0
```
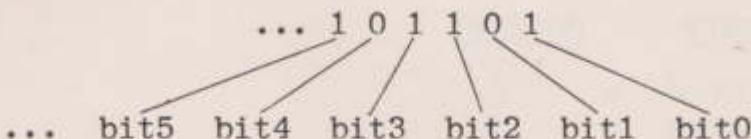
Bytes

A group of 8 bits is called a 'byte'. One 'word' on your BBC micro contains one byte or one 8-bit pattern. The entire store that is accessible to the user consists of 16,384 words or bytes on a Model A and 32,768 words or bytes on a Model B. We usually quote storage capacity in 'K' where:

$$1K = 1024 \qquad (1024 = 2^{10})$$

Because we are working on a binary system, everything is organised behind the scenes in powers of 2. Thus we say that a Model A has 16K bytes of store, i.e. 16*1024 bytes or 16*1024*8 bits.

8-bit integers

When we use a group of decimal digits to represent a non-negative integer, each digit has a weight that is a different power of 10. For example, with 5-digits:

$$7\ 3\ 9\ 2\ 4$$

weight	10000	1000	100	10	1
value	7*10000 +	3*1000 +	9*100 +	2*10 +	4*1
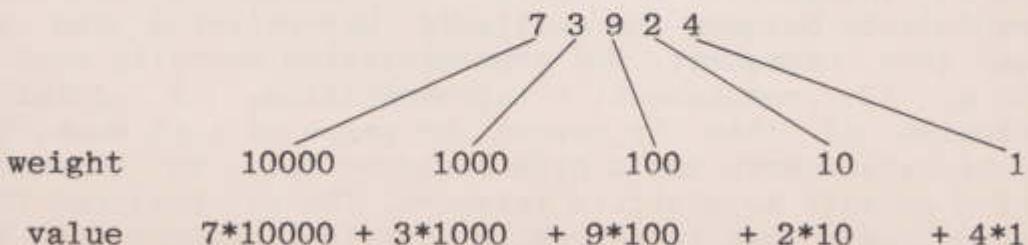
When we use a bit-pattern to represent a non-negative integer, only two values are available for each digit, so we give each digit a weight that is a power of 2. For example, with a 6-bit pattern we might have:

```
              1 0 1 1 0 1

weight    2*2*2*2*2  2*2*2*2   2*2*2    2*2      2        1
            =32        =16      =8      =4

value       1*32    + 0*16   + 1*8   + 1*4   + 0*2   + 1*1

         =   45 in decimal
```

When we use a full byte to represent an integer in this way, we have:

binary		decimal
00000000	=	0
00000001	=	1
00000010	=	2
00000011	=	3
.		.
.		.
01111110	=	126
01111111	=	127
10000000	=	128
.		.
.		.
11111110	=	254
11111111	=	255

We saw earlier that there are 256 different 8-bit patterns and they can be used in this way to represent integers in the range 0 to 255. Because it contributes least weight to an integer, the rightmost bit, bit0, is usually called the least significant bit and the leftmost bit is called the most significant.

8-bit positive and negative integers

If we want to use bytes to represent both positive and negative integers, we have to define a different correspondence between the available bit-patterns and the values they represent. The representation normally used is known as '2s complement' representation. A detailed description of this is beyond the scope of this book, but the next table shows how a byte would be used to represent negative as well as positive integers. The bit-patterns that were previously used to represent positive integers from 128

up to 255 are now used in the same order as before to represent the negative integers from -128 up to -1. In particular, -1 is represented by a bit-pattern that consists entirely of ones. This representation for negative numbers may seem rather strange, but it has many advantages when the computer is doing calculations that involve positive and negative numbers.

binary		decimal
10000000	=	-128
10000001	=	-127
10000010	=	-126
⋮		⋮
11111110	=	-2
11111111	=	-1
00000000	=	0
00000001	=	1
00000010	=	2
⋮		⋮
01111110	=	126
01111111	=	127

Note that you cannot tell by looking at a bit-pattern what sort of information it is being used to represent. This is determined by the context in which it is used and by the way it is processed by the circuits of the computer. For example, the same bit pattern might be used in different contexts to represent an integer or a character code.

Hexadecimal notation

When we are working with bit-patterns, it becomes very tedious having to write long sequences of ones and zeros when we want to specify a particular bit-pattern. We could abbreviate a byte by writing it as the equivalent positive decimal number, such as 179, but it is not at all obvious if we write 179 that we are talking about the bit-pattern 10110011. When we want to abbreviate a bit-pattern in a way that is not too far removed from its binary form, it is usual to write it in 'hexadecimal' notation (or hex for short). The bit-pattern is first divided into groups of four bits. There are 16 possible different patterns of four bits and each of these possible patterns can be represented by a single 'hexadecimal digit' as follows:

4-bit pattern	hexadecimal digit	4-bit pattern	hexadecimal digit
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

We can thus write the bit-pattern 10100011 in hex as A3:

```
10100011
  A    3
```

In BBC BASIC, we can write numbers in a program in hex if we precede the number by the symbol '&'. Thus we write &B3. Here are some other examples of bytes and the corresponding hex and decimal numbers:

byte	hex	decimal
00011111	&1F	31
00101110	&2E	46
01101001	&69	105
11111111	&FF	255

Note that &69 is quite different from decimal 69 which would be represented by the bit-pattern:

01000101 = &85

Because one hexadecimal digit corresponds to four binary digits, it is easy to visualize the bit-pattern corresponding to a hexadecimal number (provided that we are familiar with the sixteen patterns that correspond to the sixteen hex digits). Thus, for example, &B7 is easily visualized as:

```
  &B7
11010111
```

and &FA is easily visualized as:

```
  &FA
11111010
```

32-bit numbers

A numeric variable in BASIC occupies four computer words which contain four bytes or 32 bits. A number stored in such a variable is coded as a pattern of 32 bits. The way in which a 32-bit pattern is used to represent positive and negative integers is a simple extension of the 8-bit 2s complement representation introduced earlier. Note in particular that -1 is represented by a pattern of 32 ones. Details of how real numbers are coded as bit-patterns are beyond the scope of this book.

Logical operations on bit-patterns

The various logical plotting modes selected by GCOL (Chapter 9) use logical operations on bit-patterns when plotting new information on the screen. For this reason alone, some knowledge of these operations is necessary. The logical operators AND, OR, EOR and NOT treat the values to which they are applied as bit-patterns and operate on the individual bits of these patterns. A detailed knowledge of how these operations work is occasionally useful in advanced programming applications.

When a logical operation is applied to a bit-pattern or to a pair of bit-patterns, the individual bits are handled separately in creating the resultant bit-pattern. AND, OR and EOR are each applied to a pair of bit-patterns of the same length and the result is another bit-pattern of the same length. NOT is applied to a single bit-pattern and the result is another bit-pattern of the same length. We shall illustrate the behaviour of the logical operations on bytes, but they will behave in exactly the same way on shorter or longer bit-patterns.

AND

Each bit in the new pattern is the result of 'anding' the two bits in the same position in the two given bit-patterns according to the following table:

bit1	bit2	bit1 AND bit2
0	0	0
0	1	0
1	0	0
1	1	1

Thus, for example:

```
        byte1           10110100
        byte2           01100101

byte1 AND byte2     00100100
```

OR

Each bit in the new pattern is the result of 'oring' the two bits in the same position in the given bit-patterns according to the following table:

bit1	bit2	bit1 OR bit2
0	0	0
0	1	1
1	0	1
1	1	1

Thus, for example:

byte1	10110100
byte2	01100101
byte1 OR byte2	11110101

EOR

Each bit in the new pattern is the result of 'exclusive oring' the two bits in the same position in the given bit-patterns according to the following table:

bit1	bit2	bit1 EOR bit2
0	0	0
0	1	1
1	0	1
1	1	0

The name of the operator derives from the fact that it 'excludes' the case where both bits to which it is applied are 1. Thus, for example:

byte1	10110100
byte2	01100101
byte1 EOR byte2	11010001

NOT

Each bit in the new bit-pattern is the result of 'negating' the same bit in the given bit-pattern. NOT produces the 'logical inverse' of the given bit-pattern by changing 0s to 1s and 1s to 0s.

bit	NOT bit
0	1
1	0

Thus, for example:

byte	10110100
NOT byte	01001011

Representation of TRUE and FALSE

In BBC BASIC, the value TRUE is represented by a bit-pattern containing nothing but ones and FALSE is represented by a bit-pattern containing nothing but zeros. When these values are stored in numeric variables, they look like the numeric values −1 and 0.

Appendix 6 **Print formatting**

A PRINT statement contains a list of items that are to be displayed on the screen, the print list. Each item is either a numeric value or a string and the layout of these items on the screen can be controlled by using various punctuation marks in the PRINT statement.

Semicolons in the print list

When a semicolon is used to separate two items in a PRINT statement, the two items are displayed adjacent to each other with no spaces between them.

If the print list starts with a numeric value, then that first value will be treated as if it was preceded by a comma (see below) and extra spaces will appear at the start of the line. These spaces can be suppressed by inserting a semicolon before this first item.

A PRINT statement normally moves the cursor to a new line of the screen after displaying the items in the print list. The next PRINT statement obeyed would then display its items on the new line. A semicolon can be used at the end of a PRINT statement to ensure that the cursor remains on the same line so that the next PRINT statement will display its first item on the same line.

Apostrophes in the print list

The apostrophe, ', can be used anywhere between items in a PRINT statement to move the cursor to a new line before printing the next item.

Commas in the print list

In order to understand the layout produced by using commas in a PRINT statement, we must picture the screen as being divided into vertical strips or 'print fields'. These fields are usually ten characters wide, although this width can be changed by using '@%' (see below).

When a comma is used to separate two items in the print list, this causes enough extra spaces to be inserted to ensure that the item following the comma is displayed at the right hand side of the next print field. This is useful if we want items output by consecutive PRINT statements to be lined up in columns.

If the first item in a print list is numeric, and is not preceded by a semicolon, then that item will be displayed at the right of the first print field. For example,

```
10    PRINT 1,2,3
20    PRINT 2.75,5,7.4
```

will display

```
        1           2           3
        2.75        5           7.4
    _____/
        10
    characters
```

Changing the print format: @%

The value of a special variable called @% controls the format used for displaying the items output by a PRINT statement. It is best to use hexadecimal notation for giving a value to @% and for understanding how the value of @% controls the way values are printed. For example,

```
10    @% = &0102020A
```

The value of @% is a 32-bit integer which is stored as 4 bytes. We call these four bytes (from left to right) B4, B3, B2 and B1. Each byte corresponds to two hexadecimal digits (see Appendix 5) and in the above example we have B4=&01, B3=&02, B2=&02, B1=&0A.

B4 can be set to 0 or 1 and is used by the STR$ function. STR$ is used to convert a numeric value into a string and setting B4 of @% to 1 causes STR$ to use the formatting details specified by the other bytes in @% when converting a number into its string representation. Setting B4 to 0 causes STR$ to ignore the rest of @%.

Bytes B3, B2 and B1 each control a different aspect of the formatting of values output by PRINT statements.

B3 selects one of three basic formats:

value of B3	format selected
&00	General or G format: normal format in operation when machine is first switched on.
&01	Exponent or E format: numbers are always printed in scientific notation, for example: 1279.53 as 1.27953E3 0.000571 as 5.71E-4
&02	Fixed or F format: numbers always printed with a fixed number of digits after decimal point.

The effect of byte B2 depends on the basic format currently selected by B3:

basic format currently selected	effect of B2
G format	B2 gives the maximum number of digits that can be printed before reverting to E format.
E format	B2 gives the total number of digits to be printed before the E part, i.e. the accuracy to which numbers are to be printed.
F format	B2 specifies a fixed number of digits to be printed after a decimal point.

Finally B1 specifies the width of the print fields into which the screen is divided.

When the machine is first switched on, @% is set to &0000090A (or &90A). This specifies G format; numbers with more than 9 digits to be printed in E notation; the screen is divided into print fields that are ten characters wide.

To print all numbers in fixed point form with two digits after the decimal point, we can use

@% = &02020A

To print small numbers in columns that are 5 characters wide, with one digit after the decimal point, we can use

@% = &020105

Appendix 7 **Characters, ASCII codes, control codes and Teletext codes**

ASCII codes

A character is stored inside the computer as an integer that occupies 8 bits or one byte. There is an internationally agreed standard set of codes for the commonly used characters. These are the ASCII codes (American Standard Code for Information Interchange). The next table contains a list of the common visible characters together with their ASCII codes in decimal and hex.

ASCII characters and their codes

decimal code	hex code	char	decimal code	hex code	char	decimal code	hex code	char
32	&20		64	&40	@	96	&60	£
33	&21	!	65	&41	A	97	&61	a
34	&22	"	66	&42	B	98	&62	b
35	&23	#	67	&43	C	99	&63	c
36	&24	$	68	&44	D	100	&64	d
37	&25	%	69	&45	E	101	&65	e
38	&26	&	70	&46	F	102	&66	f
39	&27	'	71	&47	G	103	&67	g
40	&28	(	72	&48	H	104	&68	h
41	&29	)	73	&49	I	105	&69	i
42	&2A	*	74	&4A	J	106	&6A	j
43	&2B	+	75	&4B	K	107	&6B	k
44	&2C	,	76	&4C	L	108	&6C	l
45	&2D	-	77	&4D	M	109	&6D	m
46	&2E	.	78	&4E	N	110	&6E	n
47	&2F	/	79	&4F	O	111	&6F	o
48	&30	0	80	&50	P	112	&70	p
49	&31	1	81	&51	Q	113	&71	q
50	&32	2	82	&52	R	114	&72	r
51	&33	3	83	&53	S	115	&73	s
52	&34	4	84	&54	T	116	&74	t
53	&35	5	85	&55	U	117	&75	u
54	&36	6	86	&56	V	118	&76	v
55	&37	7	87	&57	W	119	&77	w
56	&38	8	88	&58	X	120	&78	x
57	&39	9	89	&59	Y	121	&79	y
58	&3A	:	90	&5A	Z	122	&7A	z
59	&3B	;	91	&5B	[	123	&7B	{
60	&3C	<	92	&5C	\	124	&7C	¦
61	&3D	=	93	&5D	]	125	&7D	}
62	&3E	>	94	&5E	^	126	&7E	~
63	&3F	?	95	&5F	_			

Control codes

A number of the 256 available character codes are reserved for special purposes on the BBC computer. Sending one of these codes to the display hardware by using a PRINT or a VDU statement has a special effect. These codes are usually referred to as 'VDU drivers'. Note that some of the codes must always be followed by a fixed number of additional codes or 'parameters'. If these are omitted, the next few characters printed will be taken as the missing parameters.

Summary of VDU codes

decimal	hex	parameters	effect
0	0	0	Does nothing
1	1	1	Send a character to printer only
2	2	0	Switch on printer output
3	3	0	Switch off printer output
4	4	0	Separate text and graphics cursors
5	5	0	Join text and graphics cursors
6	6	0	Enable VDU drivers
7	7	0	Beep
8	8	0	Move cursor back one space
9	9	0	Move cursor forward one space
10	&A	0	Move cursor down one line
11	&B	0	Move cursor up one line
12	&C	0	CLS (clear text screen)
13	&D	0	Move cursor to start of current line
14	&E	0	Page mode on
15	&F	0	Page mode off
16	&10	0	CLG (clear graphics screen)
17	&11	1	COLOUR c
18	&12	2	GCOL 1,c
19	&13	5	New actual colour for colour number
20	&14	0	Restore default actual colours
21	&15	0	Disable VDU drivers
22	&16	1	MODE m
23	&17	9	Create user-defined character shape
24	&18	8	Define graphics window
25	&19	5	PLOT k,x,y (2 bytes for x, 2 for y)
26	&1A	0	Restore default windows
27	&1B	0	Does nothing
28	&1C	4	Define text window
29	&1D	4	Define graphics origin
30	&1E	0	Move text cursor to top left
31	&1F	2	MOVE x,y
127	&7F	0	Backspace and delete

These codes can also be sent from the keyboard by typing a CONTROL character — hold down the CTRL key and type the character. For example, codes 1 to 26 correspond to CONTROL-A to CONTROL-Z.

Teletext control codes

In Teletext mode (MODE 7), a number of special effects can by switched on and off by displaying special control codes. Remember that one of these control codes appears as a space

on the screen and that its effect lasts only for the current screen line.

<center>Teletext control codes for MODE 7</center>

code	controls	effect
129	colour	text characters in red
130	colour	text characters in green
131	colour	text characters in yellow
132	colour	text characters in blue
133	colour	text characters in magenta
134	colour	text characters in cyan
135	colour	text characters in white
136	flash	set flashing on current line
137	flash	clear flashing on current line
140	char. ht.	single height characters
141	char. ht.	double height characters
145	graphics	graphics characters in red
146	graphics	graphics characters in green
147	graphics	graphics characters in yellow
148	graphics	graphics characters in blue
149	graphics	graphics characters in magenta
150	graphics	graphics characters in cyan
151	graphics	graphics characters in white
152	special	supress display (hide)
153	special	normal graphics (not separated)
154	special	separated graphics
156	colour	reset background colour to black
157	colour	background colour = current foreground

Teletext graphics characters

The Teletext (MODE 7) graphics characters consist of 2x3 patterns of foreground and background colour. There are two numeric codes for each of the graphics character shapes. After a line of text has been switched to graphics mode by one of the graphics codes in the previous table, the ASCII characters with codes 32 to 63 and 95 to 126 are displayed as graphics characters. (The codes from 64 to 94 are displayed as normal ASCII characters, i.e. numeric digits and capital letters.)

These ASCII codes provide a convenient way of printing a string of graphics characters. A PRINT statement in a program can contain a string of the corresponding ASCII characters, and, providing an appropriate code precedes them on the output line, they will be displayed as graphics characters.

The graphics shapes that replace the normal ASCII characters are given in the next table. Note that there is no lower code for a solid block of foreground colour.

Graphics characters that replace the normal ASCII characters

decimal code	hex code	ASCII char.	graphics char.	decimal code	hex code	ASCII char.	graphics char.
32	&20	space		95	&5F	‾	
33	&21	!		96	&60	£	
34	&22	"		97	&61	a	
35	&23	#		98	&62	b	
36	&24	$		99	&63	c	
37	&25	%		100	&64	d	
38	&26	&		101	&65	e	
39	&27	'		102	&66	f	
40	&28	(		103	&67	g	
41	&29	)		104	&68	h	
42	&2A	*		105	&69	i	
43	&2B	+		106	&6A	j	
44	&2C	,		107	&6B	k	
45	&2D	-		108	&6C	l	
46	&2E	.		109	&6D	m	
47	&2F	/		110	&6E	n	
48	&30	0		111	&6F	o	
49	&31	1		112	&70	p	
50	&32	2		113	&71	q	
51	&33	3		114	&72	r	
52	&34	4		115	&73	s	
53	&35	5		116	&74	t	
54	&36	6		117	&75	u	
55	&37	7		118	&76	v	
56	&38	8		119	&77	w	
57	&39	9		120	&78	x	
58	&3A	:		121	&79	y	
59	&3B	;		122	&7A	z	
60	&3C	<		123	&7B	{	
61	&3D	=		124	&7C	¦	
62	&3E	>		125	&7D	}	
63	&3F	?		126	&7E	~	

The other codes for graphics characters are 160 to 191 and 224 to 255. To print graphics characters using these codes, the VDU statement can be used, or CHR$ can be used to construct a string containing graphics codes. The advantage of these higher codes is that the order in which the codes correspond to the graphics shapes is more systematic. A program (or programmer) can more easily calculate a code for a given shape. We label the six cells in a graphics character as follows:

```
┌───────────┐
│ bit0 bit1 │
├───────────┤
│ bit2 bit3 │
├───────────┤
│ bit4 bit6 │
└───────────┘
```
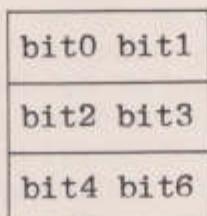
These numberings correspond to the bits in the one byte character code as follows:

$$1 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0$$

bit7 bit6 bit5 bit4 bit3 bit2 bit1 bit0
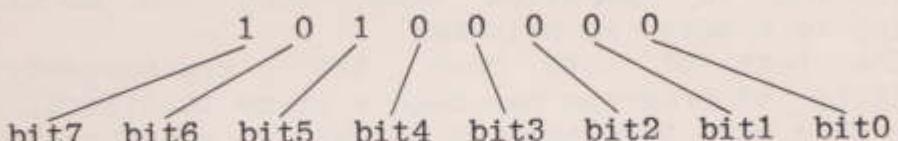
In the higher codes for graphics characters, bit5 and bit7 are always set to 1. The remaining bits are set to 1 for foreground colour and to 0 for background colour in the corresponding cell. Thus, given the bit values that specify a shape, the code for the required character can be calculated by

$$bit0 + bit1*2 + bit2*4 + bit3*8 + bit4*16 + bit6*64 + 32 + 128$$

There is no such simple expression for calculating the lower codes as you will see if you examine the Teletext design program in Section 8.5.

Appendix 8 **Notes on program efficiency**

The efficiency of a program is a loose term meaning the actual time a program takes to run. A related consideration is the amount of storage space that the program occupies either in store or on disc or cassette. With BASIC the two considerations coincide. Most of the techniques to reduce the run-time of a program also reduce its length and therefore the storage space occupied by it. We are of course referring to the space occupied by the program text and not to space demanded by data structures such as arrays.

Nowadays with the proliferation of personal computers, the run time efficiency of programs is not particularly important in most contexts. There are however situations where the run time efficiency can be critical. Animation is a case in point and the response to a large interactive program is another. In a game playing program such as chess any reduction in execution time, when the machine is responding to a move, is welcome.

In the text of this book the 'transparency' or 'readability' of programs has been a prime consideration. It is generally more important to be able to see the structure or method used, reflected in the program text. To this end we have liberally used such aids as mnemonic or long names.

To illustrate how programs can be made more efficient we start by listing an example program as it may have appeared in the text, and then alter it to make it more efficient.

```
10    MODE 4
20    INPUT xcentre, ycentre, r
30    MOVE xcentre + r, ycentre
40    FOR theta = PI/30 TO 2*PI STEP PI/30
50      xcoord = r*COS(theta)
60      ycoord = r*SIN(theta)
70      DRAW xcentre + xcoord, ycentre + ycoord
80    NEXT theta
```

The following notes itemise how various savings can be made, but we must stress before doing this that such changes should be made only if small increases in execution speed are important. The reduction in readability of the program will probably mean that you spend considerably more time getting it working. Furthermore, returning in the future to an unreadable program to further develop or modify it is a nightmare. Where efficiency is necessary, a useful approach is to keep two versions of a program - a readable version and an efficient version. (In fact the efficient version

could be generated automatically from the readable version!)
Remember that, in most cases, the programmer's time is more
important than the computer's.

Short names

Although mnemonic names substantially improve transparency,
because the program is interpreted on a character by
character basis long names carry a run time overhead. In the
more efficient version below all the variable names have
been contracted to single characters.

Unnecessary assignments

Lines 50 and 60 are technically unnecessary. Any unnecessary
assignment carries a run time overhead because it involves
an extra memory cycle. Again their inclusion enhances
transparency. In the above version of the program you can
see immediately the method being used to generate each new
coordinate value. For efficiency the arithmetic expressions
can be included in line 70 and lines 50 and 60 deleted.

```
70    DRAW xcentre + r*COS(theta), ycentre + r*SIN(theta)
```


Positioning loop independent calculations

Calculations that could be done once before entering a loop
should not be included within the loop body. There are no
examples of this in the above program, but this can be very
important in mathematical programming where complex programs
with many nested loops are common. It is easy to
inadvertently include loop independent calculations within a
loop.

Integer arithmetic

You can force the computer to perform integer arithmetic by
appending a '%' to the variable name. For example:

```
a% = 27 : b% = 537
c% = a% * b%
d% = a% DIV b%
```

will execute in approximately a third of the time it takes
to execute:

```
a = 27.356 : b = 537.421
c = a* b
d = a/b
```

This is because arithmetic with real numbers invokes extra
time overheads. Improvements that can be gained from

converting variables into integer variables (providing the context allows such a conversion) depend on many factors. For example:

```
c = a% * b
```

still uses real arithmetic because the variables are of mixed type, and the integer variable is considered to be a real variable with zero fractional part. However benefits will still result from converting as many variables in an expression to integers as is possible.

The program so far

All the above techniques applied to our original program result in:

```
10    MODE 4
20    INPUT a%, b%, r%
30    MOVE a% + r%, b%
40    s = PI/30
50    FOR t = s TO 2*PI STEP s
60      DRAW a% + r%*COS(t),  b% + r%*SIN(t)
70    NEXT t
```

Lines and spaces

Reduction in the storage space occupied by a program and further minute savings in execution speed can be obtained by eliminating all unnecessary spaces and putting as many statements as possible on each line. This, however, results in complete destruction of readability and should again be used only in a critical context where the savings are really necessary. Note that, when omitting spaces,

```
      x=y    is OK,
     x ANDy is OK,
but   xANDy  is NOT.
```

A space must be maintained when a variable is followed by a keyword. Otherwise in the above example 'xANDy' would be interpreted as a variable name.

Structural efficiency

Now the above modifications could be termed statement efficiency. Another efficiency consideration that can result in dramatic savings is structural efficiency. Is the structure of the program as efficient as it might be? For example in the circle generating program we can change:

```
        DRAW a% + r%*COS(t),   b% + r%*SIN(t)
```

to

```
        DRAW r%*COS(t), r%*SIN(t)
```

by changing line 30 to

```
    25      VDU 29, a%; b%;
    30      MOVE r%, 0
```

Altering the structure or method such that the origin is
redefined means that we can cut down on the arithmetic in
the DRAW statement by a significant amount.

    A more drastic alteration is to redefine the algorithm so
that calls to the trigonometric routines are avoided.

```
    10      MODE 4
    20      INPUT a, b, r
    30      VDU 29, a; b;
    40      MOVE r, 0
    50      x = r : y = r/10
    60      REPEAT
    70        DRAW x,y
    80          x = x - y/10 : y = y + x/10
    90      UNTIL POINT(x,y) <> 0
    100     DRAW x,y
```

The algorithm works as follows: A circle is a curve such
that, as it is traced out incrementally, x decreases and y
increases in such a way that the decrease in x is a fraction
of the current value of y, and the increase in y a fraction
of the current value of x (providing the origin is the
centre of the circle). This fraction controls the number of
times the loop is executed. It has been set equal to 10 so
that the number of times the loop is executed (and the
number of straight line segments in the circle)   is
approximately equal to that of the trigometric algorithm.

Some actual timings

The time reductions achieved by all the above techniques are
given below. This is not an unrealistic example and it
demonstrates that the most spectacular reductions   are
usually achieved by changing the algorithm. It   also
demonstrates that the more efficient algorithms tend to be
more obscure and difficult to understand — a fact of life in
computer science.

original program	3.30	secs.
statement efficiency	3.11	secs.
structural alteration (redefining origin)	3.03	secs.
structural alteration (redefining circle algorithm)	0.85	secs.

Moral

Short variable names, long lines and removal of spaces destroy the readability of a program and give very slight savings. They should be used only when timing is absolutely critical and are not generally recommended.

Appendix 9 **List of BBC BASIC keywords**

This appendix contains a complete list of BBC BASIC keywords and a very brief description of each word. The description of a word is intended to be a reminder of what the word does and is not intended to be a detailed specification of the word. Where a word is described in this book, a reference to the chapter or appendix where it is described is given in brackets. For further details, consult the User Guide.

The words are grouped together under different headings and are listed alphabetically under each heading.

Commands

AUTO: Tells the computer to invent line numbers automatically for a program that is being typed.

CHAIN: Used to LOAD and RUN a program from a file.

DELETE (Appendix 1): Deletes lines of a program.

LIST (Appendix 1): Lists all or part of a program.

LISTO (Appendix 1): Changes the layout used when a program is listed.

LOAD (Appendix 2): Loads a program from cassette file into the computer.

NEW (Introduction): Wipes out the old program and prepares for a new one.

OLD: Restores the old program if possible (for example, after a BREAK).

RENUMBER (Appendix 1): Renumbers the lines of the current program.

RUN (Introduction): Runs the current program.

SAVE (Appendix 2): Saves a program on cassette file.

TRACE: Tells the computer to print line numbers as they are obeyed.

Control statements

CALL: Used to call a machine code subroutine.

DEF (7.1): Defines a procedure or function.

ELSE (3.2): Part of an IF-THEN-ELSE statement.

END (7.1): Terminates a run of the program.

ENDPROC (7.1): Terminates a procedure and returns.

FN (7.6): Start of a function name.

FOR (4.1): Start of a FOR-NEXT loop.

GOSUB: Go to a subroutine.

GOTO (3.8): Go to a specified line.

IF (3.1, 3.2): Start of an IF-THEN or an IF-THEN-ELSE statement.

LET: Can be used to introduce an assignment statement.

NEXT (4.1): End of a FOR-NEXT loop.

ON (3.9): Start of an ON-GOTO statement.

PROC (7.1): Start of a procedure name.

REPEAT (4.4): Start of a REPEAT-UNTIL loop.

RETURN: Exit from a subroutine.

STEP (4.3): Part of a FOR statement.

STOP: Stops a program run and reports the line number.

THEN (3.1): Part of an IF statement.

TO (4.1): Part of a FOR statement.

UNTIL (4.4): End of a REPEAT-UNTIL loop.

Keyboard input/output

CLS (8.2): Clears the character screen.

COLOUR (8.2): Specifies a colour for subsequent character printing.

COUNT: A variable that records the number of characters printed since the last new line.

DATA (1.9): Specifies data to be obtained by READ statements.

GET (8.6): Waits and obtains the ASCII code of the next character typed without displaying the character on the screen.

GET$ (8.6): Waits and obtains the next character typed without displaying the character on the screen.

INKEY (8.6): Waits a specified time interval to see if a character is typed. If a character is typed within the time, returns the ASCII code of the character, otherwise returns -1. Can also be used to detect whether a particular key is being pressed at a particular instant (see User Guide).

INKEY$ (8.6): As for INKEY, but returns the character if one is typed within the time specified, otherwise it returns the empty string.

INPUT (1.1): Inputs numbers or strings from the keyboard.

POS: Finds the current horizontal position of the flashing cursor.

PRINT (1.3): Displays a given list of items on the screen.

READ (1.0): Used to obtain data from DATA statements.

RESTORE (1.9): Moves back to the start of the first DATA statement, or to a specified DATA statement.

SPC: Used in a PRINT (or INPUT) statement to print multiple spaces on the screen.

TAB (1.6): Used in a PRINT (or INPUT) statement to move the cursor to a new position.

VDU (8.1): Sends an ASCII code to the screen hardware. If the code represents a printable character, then it is displayed, otherwise it may have a special effect.

VPOS: Finds the current vertical position of the flashing cursor.

WIDTH: Used to set the overall 'page width' that the computer uses.

File input/output
BGET#: Gets a single byte from a file.

BPUT#: Puts a single byte into a file.

CLOSE# (Appendix 2): Closes a file.

EOF# (Appendix 2): Tests for the end of a file.

EXT#: Finds how large a file is (disc files only).

INPUT# (Appendix 2): Inputs items from a file.

OPENIN (Appendix 2): Opens a file for input and allocates a file number for subsequent use by the program.

OPENOUT (Appendix 2): Opens a file for output and allocates a file number for subsequent use by the program.

PRINT# (Appendix 2): Outputs items to a file.

PTR#: Used for random access to a file (disc files only).

Numeric functions and operators

ABS: Function that gives the magnitude of a number, i.e. it removes the minus sign if any.

ACS: Arc cosine function

ASN: Arc sine function.

ATN: Arc tangent function.

COS (2.4): Cosine function.

DEG: Function that converts an angle in radians into degrees.

DIV (2.3): Operator that divides two integers and discards the remainder (if any).

EXP: Function that raises the mathematical constant e (2.7183...) to a specified power.

INT (2.4): Function that converts a real value to the nearest integer value downwards.

LN: Function that calculates logarithms to the base e.

LOG: Function that calculates logarithms to the base 10.

MOD (2.3): Operator that returns the remainder after dividing two integers.

PI (2.4): Constant that has the value 3.14159265

RAD (2.4): Function that converts an angle in degrees into radians.

SGN: Function that returns the sign of a number (-1 means negative, 0 means zero, +1 means positive).

SIN (2.4): Sine function.

SQR (2.4): Square root function.

TAN (2.4): Tangent function.

Functions for string processing

ASC (8.1): Converts a character to its ASCII code.

CHR$ (8.1): Converts an ASCII code into the corresponding character.

EVAL: Evaluates an expression stored in a string.

INSTR (8.6): Tests whether one string contains another one.

LEFT$ (8.6): Picks a specified number of characters from the start of a string.

LEN (8.6): Gives the length of a string.

MID$ (8.6): Picks a specified number of characters starting at a specified point in a string.

RIGHT$ (8.6): Picks a specified number of characters from the end of a string.

STR$ (8.6): Converts a number into its string representation.

STRING$: Generates a long string that consists of a specified number of copies of a shorter string.

VAL (8.6): Generates a number from a string that contains the number in character form.

Logical operators and values

AND (3.6, Appendix 5): Logical 'and' operator.

EOR (3.6, Appendix 5): 'Exclusive or' operator.

FALSE (3.7): The logical value 'false' (represented by the value 0).

NOT (3.6, Appendix 5): Logical 'not' operator.

OR (3.6, Appendix 5): Logical 'or' operator.

TRUE (3.7): The logical value 'true' (represented by the value -1).

Graphics

CLG (9.4): Clear the graphics screen.

DRAW (1.7): Draw a line to a specified point.

GCOL (1.7, 9.4): Select a logical plotting operation and a colour.

MODE (8.2, 9.1): Switch the computer to a specified mode.

PLOT (9.3): Carry out a plotting operation to a specified point.

POINT: A function that gives the colour of a specified point on the screen.

Sound

ENVELOPE (10.7, 10.8): Used to define an envelope that can then be used for varying the amplitude and pitch of the sound produced by a SOUND statement that refers to the envelope.

SOUND (1.8, 10.2): Issues a request for a particular sound to be made by the sound generator.

Special variables

HIMEM: Contains the address of the highest memory location available for use by the user's BASIC program.

LOMEM: Contains the address of the storage location above which the computer will store all a BASIC program's

variables.

PAGE: Contains the address of the start of the area in which the user's BASIC program is or will be stored.

TIME (4.7): Contains a value that is increased by 1 every one hundredth of a second. Can be reset to any value by the user.

TOP: Contains the address of the top of the user's program.

Miscellaneous

ADVAL: Analogue to digital conversion function. Used to detect the value of one of the voltages at the analogue input socket.

CLEAR: Tells the computer to wipe out the values of all variables previously in use.

DIM (6.1) Allocates space for arrays.

ERL: Gives the number of the line at which an error last occurred.

ERR: Gives the number of the error that last occurred.

LOCAL (7.2): Used to specify that variables are local to a procedure or function.

OPT: Determines the form of output of assembly language programs.

REM (1.7): Introduces a remark or comment intended to make the meaning of a program clearer.

REPORT: Reports in words what the last error was.

USR: Calls a section of machine code program that returns a result.

Appendix 10 **Some operating system commands**

These commands are all preceded by the character *. We summarise only the commands that have been used in this book. For a complete list you should consult the User Guide.

*CAT: Used to display a list of the files stored on a tape.

*KEY: Used for defining the effect of the user-defined function keys (the red keys at the top of the keyboard — see Appendix 1).

*SPOOL: When followed by the name of a file, this command specifies that all information displayed on the screen should also be sent (in character form) to the file. The file is terminated by issuing a *SPOOL command with no file name.

*EXEC: Reads a specified file of characters and treats them as if they were being typed at the keyboard.

*FX: This command enables the user to control a large number of special effects. The effect controlled depends on the first parameter. Only the *FX calls that have been used in this book are listed.

*FX 4 is used to switch cursor editing on and off. This enables the COPY key and the cursor arrows to be used for other purposes.

*FX 4,0 switches on cursor editing.
*FX 4,1 switches off cursor editing.

When cursor editing is switched off, the editing keys generate the following ASCII codes:

COPY	135	(&87)
	136	(&88)
	137	(&89)
	138	(&8A)
	139	(&8B)

*FX 11 is used to set the time delay (in hundredths of a second) before a held-down key 'auto-repeats', i.e. sends repeated copies of the same character to the computer.

*FX 11,0 turns off the auto-repeat.

*FX 11,t sets the auto-repeat delay to t.

*FX 12   is used to set the time interval between
     successive auto-repeats of a character.

     *FX 12,0 resets auto-repeat behaviour to normal.
     *FX 12,t sets time interval between auto-repeats
              to t.

Index

Unleash your BBC micro

Your BBC Micro is waiting to amaze, amuse and delight you – and all you have to do is ask. Let this book show you how to transform your computer from an inert box of electronics into your most obedient and versatile servant.

First, the programming language BBC BASIC is described in clear and simple terms, enabling you to write your own programs at an early stage. Then, in over 150 pages, the extensive graphics animation and sound facilities on the BBC Micro are comprehensively covered. Before too long you'll be

※ inventing and playing games
※ writing and listening to music
※ producing and watching animated graphics
all in the comfort of your own front room.

A further book *Advanced Programming Techniques for the BBC Micro* by the same authors contains ideas for even more exciting programs and projects. There is a whole world waiting to be explored and the voyage of discovery begins on page one of *The BBC Micro Book: BASIC, Sound and Graphics*.

◭ Addison-Wesley Publishing Company
0-201-14058-6